
# Bachelorarbeit

**zum Thema „Sicherer Transfer und Boot von Images zur Malware Analyse aus Android Endgeräten“**

*Christopher Lenk*

*FO14w1-B*

*Hochschule Mittweida*

Mittweida, 2017

Fakultät Angewandte Computer- und Biowissenschaften

# Bachelorarbeit

**Thema: „Sicherer Transfer und Boot von Images zur Malware Analyse aus Android Endgeräten“**

Autor:
*Christopher Lenk*

Studiengang:
*Allgemeine und Digitale Forensik*

Seminargruppe:
*FO14w1-B*

Erstprüfer:
*Prof. Dr. rer. nat. Christian Hummert*

Zweitprüfer:
*Prof. Dr. rer. pol. Dirk Pawlaszczyk*

Einreichung:
*Mittweida, den 14. Juli 2017*

Verteidigung/Bewertung:
*Mittweida, 2017*

Faculty Applied Computer Sciences & Biosciences

# Bachelor Thesis

**topic: „Secure transfer and boot of images for malware analysis from Android phones“**


author:
*Christopher Lenk*

course of studies:
*General and Digital Forensic Science*

seminar group:
*FO14w1-B*

first examiner:
*Prof. Dr. rer. nat. Christian Hummert*

second examiner:
*Prof. Dr. rer. pol. Dirk Pawlaszczyk*



submission:
*Mittweida, 14.07.2017*

defense/evaluation:
*Mittweida, 2017*

# Bibliographische Beschreibung

Lenk, Christopher:
Sicherer Transfer und Boot von Images zur Malware Analyse aus Android Endgeräten. - 2017. - X, 42, 1 S.
Mittweida, Hochschule Mittweida, Fakultät Angewandte Computer- und Biowissenschaften, Bachelorarbeit, 2017.

# Referat

Das Smartphone ist im Laufe seiner Entwicklung zu einem ständigen Begleiter des modernen Menschen avanciert. Besonders der Marktführer, das mit Android betriebene Mobilgerät, ist weit verbreitet, allerdings aufgrund von nicht geschlossenen Sicherheitslecks für Malware anfällig, sodass zahlreiche Arten an Android-Viren existieren. Die Analyse dieser ermöglicht die Klassifikation, Abwehr und Bekämpfung solcher Angriffe. Die Kombination aus einem Webserver und mehreren Wandboards, einer Mikrocontroller-Entwicklungsplattform, bietet zusätzlich die Möglichkeit, das Analysieren zur Laufzeit durchzuführen. Im aktuellen Zustand des geplanten Projektes wäre es allerdings nötig, bei jedem Wechsel der Android-Version oder -Malware die bootfähige Speicherkarte des Wandboards auszutauschen. Dieser zeitlich und materiell aufwändige Vorgang kann durch die Konfiguration des Wandboards, das heißt seiner Hard- und Software, für den Boot der Android-Abbilder mit den zu analysierenden Viren über das Netzwerk optimiert werden. Die vorgestellte Methodik nutzt das Protokoll Preboot Execution Environment, um die Android-Systemdateien über einen TFTP- und einen NFS-Server zu laden. Die SD-Karte wird ausschließlich für den verwendeten Bootloader U-Boot eingesetzt. Um die auf den Servern eingerichteten Daten herzustellen, wurden zwei Ansätze, die Kompilierung des Android-Quellcodes und die Generierung aus den vorinstallierten SD-Karten-Images, entwickelt. Das Vorgehen zur Konfiguration der Server und des Bootloaders ist als Anleitung in die methodische Darstellung eingearbeitet. Mit Abschluss der Einrichtung zum Netzwerkstart konnte Android erfolgreich auf dem Wandboard über das lokale Netz gestartet werden. Die nunmehr mit weniger Aufwand durchführbaren Wechsel von Systemversionen ermöglichen die Realisierung eines Services zur Android-Malware-Analyse zur Laufzeit, der die zeitnahe und sichere Klassifikation und Bekämpfung von Viren auf Mobilgeräten möglich macht.

# Abstract

In the course of its development the smartphone has become a constant companion of modern human. Especially the market leader, the Android-powered mobile device, is widespread, but due to non-closed security leaks it is susceptible to malware, so there are many types of Android viruses. Their analysis allows the classification, defense and control of such attacks. A combination of a web server and several Wandboards, a microcontroller development platform, also offers the possibility to perform the analysis at runtime. In the current state of the planned project it would be necessary to exchange the bootable memory card of the Wandboard with every change of the Android version or malware. This time-consuming and materially complex process can be optimized through the configuration of the Wandboard, its hardware and software to boot the Android images with the viruses over the network. The presented methodology uses the Preboot Execution Environment protocol to load the Android system files via a TFTP and a NFS server. The SD card is only used for the bootloader U-Boot. In order to create the data set up on the servers, two approaches, the compilation of the Android source code and the generation from the pre-installed SD card images, were developed. The procedure for configuring the servers and the bootloader has been incorporated into the methodical presentation as a guide. With completion of the network boot setup, Android was successfully launched on the Wandboard over the LAN. The change of system versions, which can be carried out with less effort now, makes it possible to implement an Android malware analysis service at runtime, which enables the timely and secure classification and combat of viruses on mobile devices.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Abkürzungsverzeichnis

**AOSP** Android Open Source Project

**ARM** Advanced RISC Machines

**ASCII** American Standard Code for Information Interchange

**BDM** Background Debug Mode

**BOOTP** Bootstrap Protocol

**CPU** Central Processing Unit

**CRC32** Cyclic Redundancy Check

**DDR3-SDRAM** (kurz DDR3) Double Data Rate Synchronous Dynamic Random Access Memory (Version 3)

**DHCP** Dynamic Host Configuration Protocol

**EDM** Embedded Design Module

**GB** Gigabyte

**HDMI** High Definition Multimedia Interface

**$I^2C$** Inter-Integrated Circuit

**IP** Internet Protocol

**JTAG** Joint Test Action Group

**KiB** Kibibyte

**LAN** Local Area Network

**MAC** Media Access Control

**MB** Megabyte

**MIPS** Microprocessor without interlocked pipeline stages

**MIT** Massachusetts Institute of Technology

**NFS** Network File System

**NOR** Not OR - Nicht oder

**OpenJDK** Open Java Development Kit

**PowerPC** Performance optimization with enhanced RISC Performance Chip

**PXE** Preboot Execution Environment

**RARP** Reverse Address Resolution Protocol

**RFC** Request For Comments

**RISC** Reduced Instruction Set Computer

**ROM** Read-Only Memory

**SATA** Serial Advanced Technology Attachment

**SD** Secure Digital Memory Card

**SPI** Serial Peripheral Interface

**SPL** Secondary Program Loader

**TFTP** Trivial File Transfer Protocol

**U-Boot** Universal Bootloader

**UDP** User Datagram Protocol

**USB** Universal Serial Bus

**USB OTG** USB On-The-Go

**WB** Wandboard

**WLAN** Wireless Local Area Network

# 1 Einleitung

Mit der Entwicklung des ersten Smartphones Ende der 90er Jahre des letzten Jahrhunderts brach eine neue Ära in der Kommunikation zwischen Menschen an. Das klassische Mobiltelefon wurde im Laufe der Entwicklung durch Funktionen erweitert, die zuvor nur Computer, Navigationsgeräte, Digitalkameras oder Media Player zur Verfügung stellen konnten, sodass ein neuartiger PDA (Personal Digital Assistant) mit der Möglichkeit zur Telefonie entstand. Der Touchscreen digitalisierte das klassische Tastenfeld und vielzählige Apps und die Internetfunktion brachten neue Nutzungsmöglichkeiten vom mobilen Kalender über Nachrichten-Widgets bis hin zu zahlreichen Spiele-Applikationen. Die Verbindung von Telefon und Internet erweitert den Alltag des modernen Menschen: Er kann unterwegs auf Webseiten zugreifen, seine E-Mails abrufen oder über Chatprogramme mit Freunden in Kontakt treten. Aufgrund der breiten Möglichkeiten und Erleichterungen haben die Smartphones der verschiedenen Hersteller einen großen Nutzerkreis und verbinden zahlreiche Menschen.
Die Anbindung ans Internet ist allerdings Fluch und Segen zugleich. Wegen der weiten Verbreitung und der enthaltenen sensiblen Daten erweiterte sich dadurch auch der Zielbereich krimineller Angriffe zunehmend auf das Smartphone und dessen Betriebssystem.
Android, der Marktführer unter den Smartphone-Betriebssystemen, wurde unter der Federführung von Google entwickelt und hat sich seit der Veröffentlichung 2008 einen Marktanteil von 81,7 % im vierten Quartal 2016 erarbeitet. Der Großteil des restlichen Anteils wird mit 17,9 % vom System iOS des Herstellers Apple eingenommen. (Ga17) Aufgrund ihrer Verbreitung in der Smartphone-Welt sind beide Betriebssysteme häufig Angriffen ausgesetzt, besonders durch Malware. Da Apple bei iOS regelmäßig Sicherheitsrisiken schließt und nur Apps aus dem AppStore zulässt, ist ein erfolgreicher und lohnenswerter Angriff eher schwierig. Android hingegen besitzt einige Sicherheitslücken im System und in einigen Apps, die noch nicht vollständig geschlossen wurden, sodass ein Angreifer diese für seine Zwecke ausnutzen kann. Bei Erfolg einer Attacke werden aufgrund des Marktanteils eine große Zahl an Smartphone-Nutzern und deren Daten verwundet. Für einen Virus ist Android damit ein lohnendes Ziel.
Malware kann allerdings durch spezifische Programmausführungen beziehungsweise Verhaltensweisen erkannt und klassifiziert werden. Mithilfe eines simulierten Android-Smartphones könnte man daher etwa einen bestimmten Virus anhand seiner Vorgehensweise und der dadurch entstehenden Veränderungen am System beobachten und analysieren. Allerdings erkennen die meisten Virus-Arten, ob sie auf einem realen oder simulierten Gerät ausgeführt

werden und können ihr Verhalten bei einer Simulation verfälschen. Um das zu verhindern, kann eine Entwicklungsplattform verwendet werden, die Wandboard genannt wird. Mit dieser ist es möglich, eine Smartphone-ähnliche, aber nicht komplett simulierte Umgebung zu schaffen, in der eine fehlerfreie Malware-Analyse durchgeführt werden kann.
Das beschriebene Vorhaben ist als Projekt bereits durch eine Master- und eine Bachelorarbeit untersucht und realisiert worden.
Die Konfiguration und Einrichtung des Wandboards wurde in einem dieser Arbeit vorangehenden Bachelorprojekt durchgeführt. Die vorgestellte Methodik umschreibt zunächst die Einrichtung des grundlegenden Android-Systems auf der SD-Karte. Die folgende Etablierung der Android Debug Bridge (ADB) ermöglichte die weiteren Konfigurationen am Dateisystem. Mit dem erfolgreichen Einhängen der Ordnerstruktur war es möglich, den Play Store und andere Anwendungen zu installieren und einzurichten. Nach der Erläuterung der Inbetriebnahme von schadhaften Applikationen in unterschiedlichen Varianten konnten erste Beobachtungen und Analysen durchgeführt und ausgewertet werden. Das Verhalten der aktivierten Schadsoftware wurde dabei mit der Laufzeit protokolliert, die Veränderungen, die sie am System vorgenommen hatte, können mithilfe der gesammelten Daten nachvollzogen werden. Ein Bericht dokumentiert das Geschehen und gibt dem Analytiker schließlich alle gewonnenen Informationen aus. (Mo16)
Die zeitlich nach der soeben beschriebenen Bachelorausarbeitung folgende Masterarbeit setzte die Erkenntnisse dieser zur Realisierung einer Plattform zur Malware-Beobachtung und -Analyse fort. Dabei erfolgte die Installation zusätzlicher Dienstprogramme sowie die Einrichtung von Skripten, die die Systemüberwachung und Datensammlung übernehmen. Deren Ergebnisse werden mit Ende des programmatischen Durchlaufs auf dem Host-System abgelegt. Zudem erfolgte die Einbindung bisheriger Fortschritte in ein größeres Projekt zur Malware-Analyse auf dem Wandboard, ergänzt durch einen Webserver mit entsprechender Webseite zum Hochladen der Malware und Auswahl der gewünschten Hardware-Version. (Wi16)
Die aktuell noch offene Problemstellung umfasst die Installation der Android-Abbilder mit dem zu testenden Virus auf dem Gerät. Im momentanen Zustand ist für jedes neue Abbild die Bereitstellung einer neuen SD-Karte notwendig, welche mit jedem Wechsel der Betriebssysteme und Viren ausgetauscht werden muss. Dieser sowohl zeitlich als auch materiell aufwendige Prozess kann durch das automatisierte Laden von Test-Images über das Netzwerk vereinfacht werden.
Die Beschreibung und Konfiguration der Automatisierung ist Ziel dieser Arbeit. Dabei soll das Wandboard ein neues Systemabbild über das Netzwerk laden können. Diese Funktionalität bietet das Netzwerkprotokoll Preboot

Execution Environment. Da das Wandboard die Grundvoraussetzungen für PXE erfüllt, soll es im Zuge der Durchführung für die Realisierung des Protokolls eingerichtet werden. Dabei soll der für ARM-Architekturen nutzbare Bootloader U-Boot Anwendung finden, der PXE unterstützt und das Booten über das Netzwerk auf dem Wandboard ermöglicht. Das Protokoll wird dabei durch einen Network File System-Server ergänzt, der das Android-Dateisystem enthält und Änderungen abspeichern kann.
Die vorliegende Arbeit gliedert sich in vier Abschnitte. Um die Ausführungen der angewandten Methodik in vollem Umfang zu erfassen, ist zuvor das Verständnis einiger Grundlagen notwendig. Zunächst erfolgt die Erläuterung der Hardwareplattform Wandboard. Dabei wird auf den Aufbau und die Einsatzmöglichkeiten in Bezug auf Funktionalität und Betriebssysteme eingegangen und die grundlegende Einrichtung erläutert. Weiterhin soll der verwendete universelle Bootloader, das U-Boot, näher betrachtet werden, besonders im Hinblick auf den zur Verfügung stehenden Befehlsumfang und dessen Anwendbarkeit in der Anpassung des Bootprozesses. Das Grundlagenkapitel wird abgeschlossen durch einen Einblick in die beim Netzwerk-Boot beteiligten Protokolle. Dabei soll vor allem der Ablauf der Verbindungsetablierung und Datenübermittlung im Vordergrund stehen. Das Unterkapitel geht auf die Protokolle PXE, das in diesem verwendete TFTP und das ergänzende NFS ein, die in der Durchführung des angestrebten Bootvorgangs eine entscheidende Rolle übernehmen.
Nachfolgend wird in der Methodik das genaue Vorgehen bei der Einrichtung und Konfiguration des U-Boot- und Android-Systems für das Netzwerk-Booten auf dem Wandboard dargestellt. Zunächst erfolgt die Darlegung vorbereitender Schritte zur Erstellung der benötigten Kernel- und Betriebssystem-Dateien, bevor die eigentliche Realisierung der Funktion des Netzwerkstarts beginnt. Die dargelegten Ansätze sind als Anleitung angelegt und sollen anderen Nutzern den Weg zu einer erfolgreichen Konfiguration erleichtern.
Im Ergebnisteil wird daraufhin der konkrete Verlauf des Hochfahrens über das lokale Netz präsentiert. Es erfolgt eine Analyse anhand der Resultate der Konfigurationen und der zeitlichen Dauer.
Mit der Diskussion schließt diese Arbeit und ermöglicht eine Verbindung und Auswertung der gemachten Zielsetzung mit den erreichten Ergebnissen. Möglichkeiten zur weiteren Entwicklung des Projektes werden abschließend aufgezeigt.

# 2 Grundlagen

## 2.1 Wandboard

Die Entwicklungsplattform Wandboard ist als Einplatinencomputer realisiert, der auf der CPU-Architektur ARM-Cortex-A9 basiert. Der Prozessor vom Typ NXP i.MX6 mit einem Gigahertz Leistung ist je nach Modell mit einem Kern (WB Solo) beziehungsweise zwei (WB Dual) oder vier Kernen (WB Quad) ausgestattet.

### 2.1.1 Aufbau

Das Wandboard besteht aus zwei Modulen: Schnittstellenmodul und Kernmodul sind über den EDM-Standard miteinander verbunden.

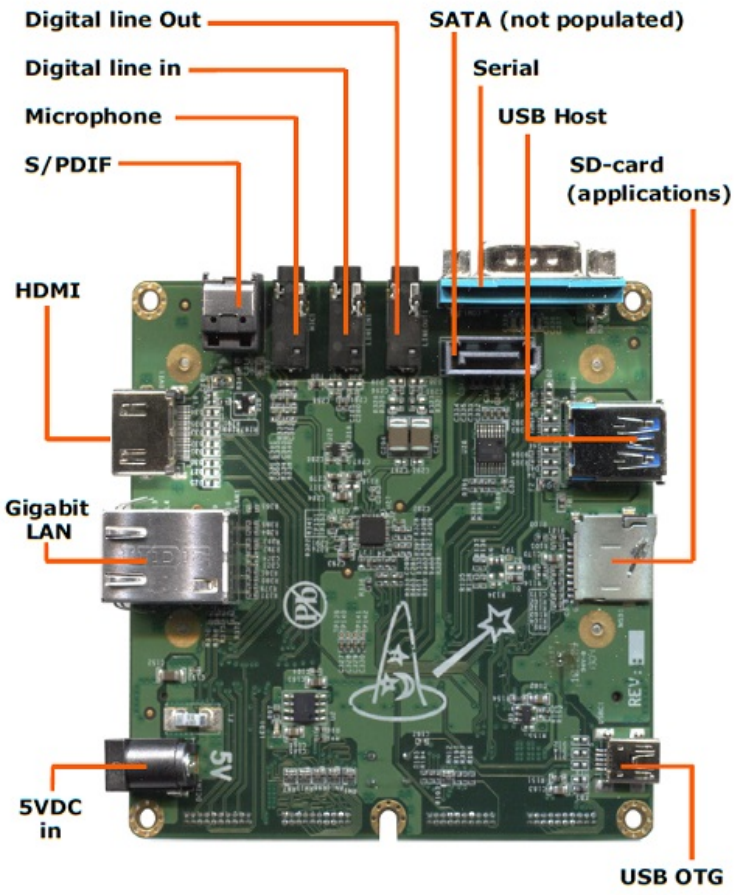


Abbildung 1: Aufbau des Schnittstellenmoduls
(files.linuxgizmos.com/wandboard-details-top.jpg)

Wie der Name bereits verrät, beinhaltet das Schnittstellenmodul, zu sehen in Abbildung 1, die Anschlüsse des Wandboards. Die Plattform verfügt über Gigabit LAN, einen HDMI-Anschluss, Audio-Video-Zugänge, einen SATA-Anschluss sowie einen SD-Karten-Slot für Anwendungen. Außerdem sind hier die 5V-Stromversorgungsbuchse, ein USB- und ein USB OTG-Anschluss zur Verbindung mit anderen USB-Geräten zu finden.

An die Unterseite des Schnittstellenmoduls ist das Kernmodul verschraubt. Unterhalb des verbindenden EDM-Standards ist der Reset-Button angebracht, der einen Neustart durchführen kann. Hier befinden sich auch die Expansion Pins, die Funktionen für Input/Output ($I^2C$, SPI, etc.) bereitstellen.

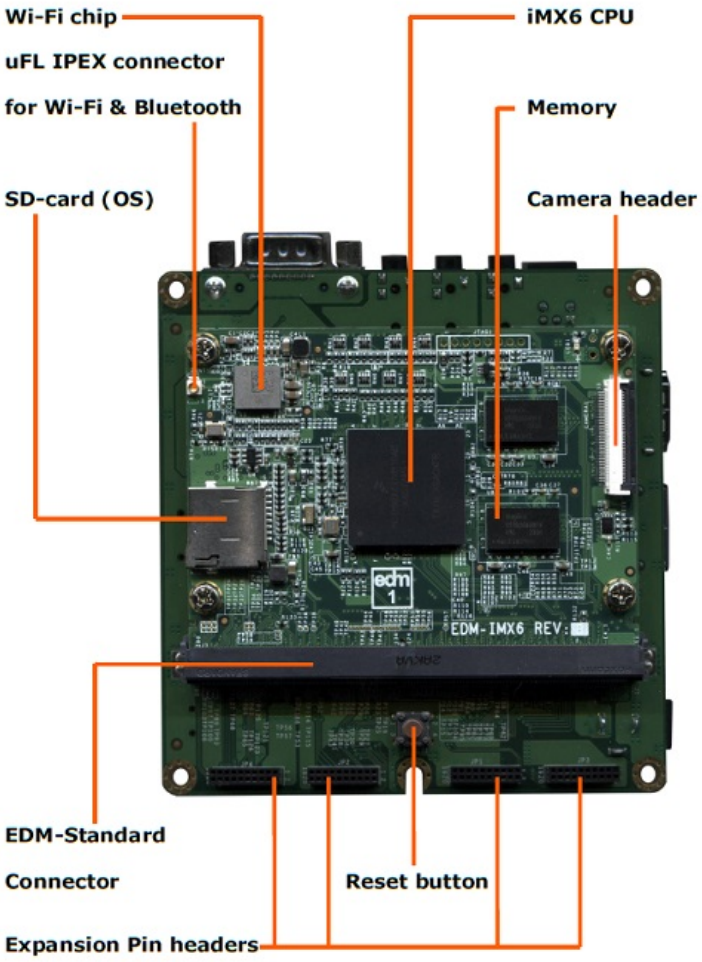


Abbildung 2: Aufbau des Kernmoduls
(files.linuxgizmos.com/wandboard-details-bottom.jpg)

Mittig auf dem Kernmodul (Abb. 2) befinden sich das Herz des Wandboards, die i.MX6 CPU und die DDR3-Speicherchips. Je nach Modell sind entweder 512 MB (WB Solo), 1 GB (WB Dual) oder 2 GB (WB Quad) Arbeitsspeicher verfügbar.
Für das Betriebssystem existiert auf dieser Ebene ein zusätzlicher Slot für die bootfähige SD-Karte. Zudem sind hier der Chip und die Schnittstelle für WLAN- (802.11n) und Bluetooth-Verbindungen sowie der Anschluss für die Wandboard-Kamera verbaut. Das Kernmodul ist von einem Kühlgitter bedeckt (in Abbildung 2 aus Sichtgründen entfernt), das die von den Chips erzeugte Wärme nach außen ableitet.

### 2.1.2 Funktion und Betriebssysteme

Für das Wandboard existieren zahlreiche Betriebssystem-Versionen, die speziell für die Funktion auf der Entwicklungsplattform entworfen wurden. Dabei bietet sich ein breiter Fächer an ausführbaren Systemen vom Smartphone-Betriebssystem Android über Ubuntu und Debian bis hin zu OpenSUSE, Fedora und FreeBSD.

Tabelle 1: Für das Wandboard verfügbare Betriebssystem-Versionen

| System | Versionen |
|---|---|
| Android | 6.0 Marshmallow<br>5.0 Lollipop<br>4.4 Kitkat<br>4.1, 4.2, 4.3 Jellybean |
| Ubuntu | 16.04 Xenial Xerus<br>15.10 Wily Werewolf<br>15.04 Vivid Vervet<br>14.04 Trusty Tahr<br>12.04 Precise Pangolin<br>11.10 Oneiric Ocelot |
| Debian | 8 Jessie |
| FreeBSD | 11.0 (nur Wandboard Quad) |
| OpenSUSE | 13.2 Harlequin<br>12.3 Darthmouth |
| Weitere | Yocto 1.4, 1.5, 1.7<br>Fedora 22<br>OpenMandriva LX 2015<br>OpenWRT 14.07 |

(Wa17)

Die in Tabelle 1 zu sehende große Vielfalt an lauffähigen Versionen macht das Wandboard zu einem vielseitig einsetzbaren Werkzeug für Entwickler. Damit ist es beispielsweise möglich, für verschiedene Systeme Programme oder Apps zu schaffen und zu testen.

Um das Wandboard auf normalem Wege erfolgreich zu starten, muss das gewählte Betriebssystem auf eine Micro-SD-Karte gebracht werden. Da die Abbilder bereits als Datenträger-Images herunterladbar sind, reicht es aus, diese über Tools wie Win32DiskImager oder den Konsolenbefehl `dd` auf die SD zu schreiben und die Speicherkarte in den dafür vorgesehenen Slot im Kernmodul zu bringen. Danach kann das System auf dem Wandboard bereits hochgefahren werden, nachdem Monitor, Peripherie und Stromanschluss angebracht wurden.

## 2.2 Das U-Boot - the Universal Bootloader

Das U-Boot steht als Abkürzung für einen universellen Bootloader, der auf verschiedenen Prozessortypen arbeitsfähig ist und von der DENX Software Engineering in der Programmiersprache C entwickelt wurde. Da dieser unter anderem auch für die Architekturen der Mikrocontroller wie PowerPC, ARM und MIPS einsetzbar ist, kann er ebenfalls als Bootloader für das Wandboard verwendet werden und diesem zahlreiche Funktionen zur Verfügung stellen, die in der Normalausführung beziehungsweise mit anderen Betriebssystemen nicht möglich wären.

### 2.2.1 Technischer Hintergrund

Das Projekt ist Open Source, das heißt innerhalb der Lizenz uneingeschränkt nutzbar, veränderbar und über die Plattform Git unter http://git.denx.de/ frei verfügbar. Die Wandboard-Version des Smartphone-Betriebssystems Android ist zur Auslieferung bereits mit U-Boot ausgestattet und bootet automatisch über diesen. Die restlichen verfügbaren Distributionen für das Wandboard besitzen den Bootloader nicht und starten über eigene Entwicklungen. Bei diesen müsste eine nachträgliche Installation durchgeführt werden.
Das U-Boot belegt bei einer normalen Konfiguration einen Speicherplatz von 128 KiB im NOR Flash Memory. Es unterstützt Geräte mit 128 KiB ROM oder mit 256 KiB NOR.
U-Boot wurde nach dem Prinzip entwickelt, „so schnell wie möglich“(De17) zu sein, um dem Nutzer ohne großen zeitlichen Aufwand ein laufendes Betriebssystem zur Verfügung zu stellen. Dazu werden einerseits Caches, also

Zwischenspeicher, eingesetzt, andererseits aktiviert U-Boot bestimmte Schnittstellen und Geräte erst, wenn sie wirklich benötigt werden.
Der Bootloader gibt den Beginn und Abschluss aller Vorgänge über die Konsole an den Nutzer weiter, um diesem die Möglichkeit zu geben, den Prozess nachzuvollziehen und eventuelle Fehler einem Arbeitsschritt zuordnen zu können. Dafür wird eine serielle Verbindung benötigt. U-Boot kann weiterhin über JTAG oder BDM in den Debug-Modus versetzt werden. (De17)

### 2.2.2 Funktionsumfang

Neben der Universalität in Bezug auf die Prozessoren bietet das U-Boot verschiedene Arten des Boot-Vorgangs und ermöglicht über Kommandozeilen-Befehle Speichermodifikationen. Das Boot-Verhalten ist flexibel an die Bedürfnisse des Nutzers anpassbar, sodass U-Boot auch für sonst schwierig zu realisierende Vorgänge Möglichkeiten bietet.
Nachfolgend soll die Zusammenstellung der verfügbaren U-Boot-Befehle eine Übersicht über die Anpassungsmöglichkeiten bieten.

**Informationsbefehle** Die Befehle in Tabelle 2 sind grundlegend zur Anzeige beziehungsweise Auflistung von Informationen über das Wandboard, seine Geräte und den Speicher einsetzbar. Zudem kann die Hilfe genutzt werden, um Auskunft über U-Boot und seine Befehle zu erhalten.

Tabelle 2: Übersicht Informationsbefehle

| Befehl | Funktion |
|---|---|
| bdinfo | Gibt die gesammelten Daten über das Board aus (zum Beispiel Speicheradressen, Frequenzen, MAC- und IP-Adresse). |
| coninfo | Zeigt Informationen über verfügbare I/O-Geräte an. |
| flinfo | Ermöglicht die Anzeige von Informationen über den Flash-Speicher. |
| iminfo *addr* | Wertet den Header eines Application-Images (Linux Kernel oder RAM-Disk) aus (Name, Typ, Größe), der an der Speicheradresse *addr* beginnt, und kontrolliert die CRC32-Prüfsumme. |
| help [*command*] | Gibt eine Liste aller möglichen Befehle aus. Wird als Parameter ein Befehl übergeben, erhält man den Hilfetext für diesen. |

(http://www.denx.de/wiki/view/DULG/UBootCmdGroupInfo)

**Speichermodifikation** U-Boot ist mit den in Tabelle 3 gezeigten Befehlen unter anderem zum Kopieren, Vergleichen und Anzeigen von Speicherbereichen fähig. Damit ist das Tool nicht nur ein Bootloader, sondern ähnelt auch einem Hex-Editor, der zur Ansicht und Modifikation von Speicher verwendbar ist.

Tabelle 3: Übersicht Speicherbefehle

| Befehl | Funktion |
|---|---|
| base [*off*] | Gibt den Basis-Offset für alle Speicher-Befehle aus. Default-Wert ist 0. Ein neuer Offset als Parameter *off* dient dem nächsten Befehl als Beginn-Adresse. |
| crc32 [*von*][*bis*] | Berechnet die CRC32-Prüfsumme über den gegebenen Speicherbereich. Als dritter Parameter kann ein Offset festgelegt werden, an welchen das Ergebnis geschrieben wird. |
| cmp [.b, .w, .l] *addr1 addr2 count* | Vergleicht den Inhalt zweier Speicher-Bereiche *addr1* und *addr2* auf Gleichheit. Mit *count* kann die Länge des Bereiches festgelegt werden, sonst stoppt der Test am ersten Unterschied. Speicherzugriff mit 8 Bit (.b), 16 Bit (.w) oder 32 Bit (.l) möglich. |
| cp [.b, .w, .l] *source target count* | Kopiert einen Speicherbereich in der Größe von *count* von *source* nach *target*. |
| md [.b, .w, .l] *address* [*count*] | Gibt den Speicherbereich am Offset *address* aus. Mit *count* lässt sich der Anzeigebereich eingrenzen. |
| mm [.b, .w, .l] *address* | Ermöglicht die Modifizierung von Speicherinhalten. Diese erfolgt schrittweise in Blöcken je nach gewähltem Speicherzugriff über die Eingabe des Nutzers an der Konsole. Handelt es sich um eine Hexadezimalzahl, wird der betreffende Bereich überschrieben. |
| mw [.b, .w, .l] *address value* [*count*] | Initialisiert den Speicherplatz am Offset *address* mit dem übergebenen Wert. Ist die *count*-Variable vorhanden, wird der gesamte angegebene Bereich mit *value* beschrieben. |

(http://www.denx.de/wiki/view/DULG/UBootCmdGroupMemory)

**Ausführungs- und Download-Kommandos** Trotz zahlreicher erweiterter Funktionen bleibt das Booten des Betriebssystems Hauptaufgabe des

Bootloaders. Daher ist es notwendig, dass auch das U-Boot einen Befehlssatz zur Ausführung verschiedener Formen von System-Images besitzt.
Dazu gehört ebenfalls die Fähigkeit, Abbilder über das Netzwerk zu booten. Die für dieses Projekt wichtigste Funktion liegt im TFTP-Support des Bootloaders. Über diesen ist es möglich, ein Datenträger-Image über das Netzwerk und PXE von einem TFTP-Server herunterzuladen und auszuführen. Das U-Boot besitzt mehrere Variablen und Befehle, mit denen ein Netzwerkboot übernommen werden kann. (DM17) (Ha11)

Tabelle 4: Übersicht Download- und Boot-Befehle

| Befehl | Funktion |
|---|---|
| source [*addr*] | Führt ein Skript-Image, ein Shell-Skript in U-Boot-Format, aus. |
| bootm [*addr*[*arg*]] | Bootet ein Betriebssystem-Abbild, das an der übergebenen Speicheradresse *addr* liegt. |
| go *addr* [*arg*] | Startet eine sogenannte Standalone-Application, die kein Betriebssystem benötigt. |
| setenv *var* [*value*] | Erstellt die U-Boot-Variable *var* mit dem übergebenen Wert *value*. Ohne Wertparameter wird die angegebene Variable gelöscht. |
| saveenv / env save printenv [*var*] | Speichert alle zuvor gemachten Parameter-Festlegungen. Die Anzeige der geschriebenen Variablen erfolgt über printenv, die Übergabe des Namens *var* beschränkt die Ausgabe. |
| run *var* | Führt die in einer Variable *var* gespeicherten Befehle in der U-Boot-Umgebung aus. |
| dhcp | Aufruf des DHCP-Clients. |
| bootp *loadAddr* [[*hostIP:*]*bootfilename*] | Netzwerkboot über BOOTP und TFTP. |
| tftpboot *loadAddr* [[*hostIP:*]*bootfilename*] | Netzwerkboot über TFTP. |

(http://www.denx.de/wiki/view/DULG/UBootCmdGroupExec und .../UBootCmdGroupDownload sowie .../UBootCmdGroupEnvironment)

## 2.3 Einführung in die Protokolle des Netzwerkstarts

Mit diesem Kapitel soll ein Einblick in die Welt der Netzwerk-Protokolle PXE, TFTP und NFS ermöglicht werden, die bei der Durchführung eines Betriebssystemstarts über das Internet beteiligt sind und dabei ihre Funktionalitäten zur Verfügung stellen.

### 2.3.1 Preboot Execution Environment

Das Protokoll Preboot Execution Environment (PXE) wurde am 20. September 1999 von Intel eingeführt und ermöglicht es, ein System ohne bootfähige Festplatte über LAN mit dem gewünschten Betriebssystem von einem entfernten Speicherort zu booten.
PXE basiert auf den Netzwerk-Protokollen DHCP und TFTP, die die Netzkonfiguration in Bezug auf die Adresszuweisung und die Übermittlung der Boot-Dateien übernehmen.
Ein Client möchte im Beispielfall nun ein Betriebssystem laden, das sich auf einem physisch getrennten Boot-Server befindet. Er sendet im ersten Schritt zunächst ein DHCP Discover an den Port 67 eines DHCP-Servers. Die Informationen des PXE-Clients wie Identifier und Art der System-Architektur werden über die DHCP Extension Tags an den Server übermittelt.
Der DHCP-Service antwortet mit einem erweiterten DHCP Offer an Port 68 (Reply Port). Der Client erhält eine IP-Adresse und die Server-Informationen werden wiederum über die Extension Tags versendet. In der PXE-Erweiterung des DHCP Offer ist eine Liste mit möglichen Boot-Servern enthalten, aus denen der Client später wählen kann.
Um die Standardprozedur des Protokolls abzuschließen, folgt die Annahme eines Angebots über den DHCP Request, den der Client erneut an Port 67 des Servers sendet. Mit der Bestätigung auf Server-Seite durch DHCP Ack (Acknowledgement) an Port 68 ist die Initialisierung der Verbindung abgeschlossen (Abb. 3).

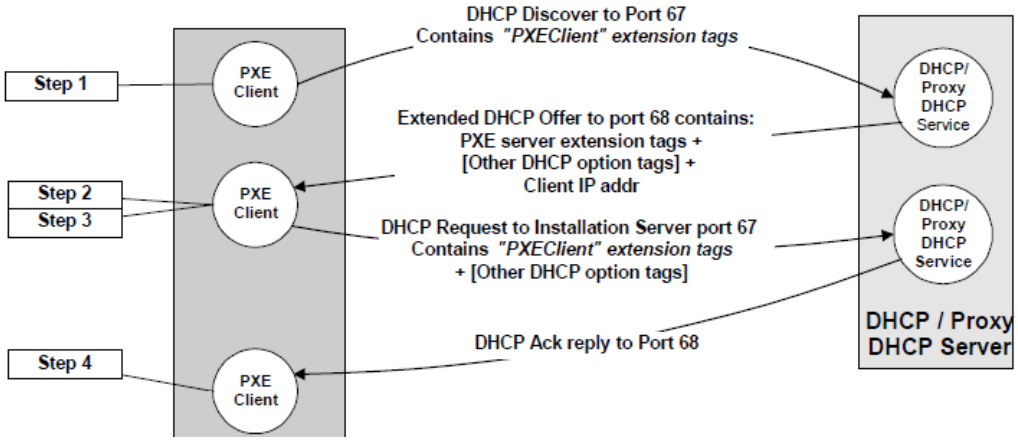


Abbildung 3: Herstellung der DHCP-Verbindung
(In99, Figure 2-1)

Nun kann der Client aus der Liste möglicher Boot-Server wählen. An den gewünschten Service sendet er ein DHCP Discover mit seiner IP-Adresse, den

Informationen wie in Schritt eins und dem Typ des Boot-Servers. Dieses Paket kann entweder als Broadcast an Port 67 oder als Multi- beziehungsweise Unicast an Port 4011 übertragen werden, je nachdem was in den Optionen des DHCP Offers des PXE Services eingetragen war.
Der Boot-Service antwortet mit einer DHCP Ack Nachricht an den Source Port des Clients. Diese enthält neben Konfigurationsparametern den Namen der Boot-Datei, die über das Protokoll TFTP heruntergeladen werden kann. Dazu erfolgt eine Anfrage an den TFTP-Service des Boot-Servers über den Standard-TFTP-Port 69. Die ausführbare Datei wird daraufhin auf das Client-System heruntergeladen und im Speicher platziert (Abb. 4).

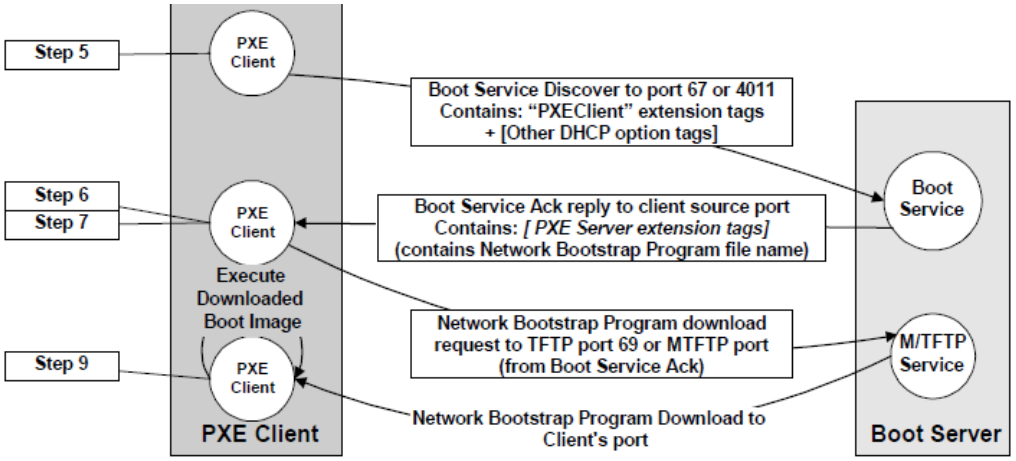


Abbildung 4: Kommunikation mit dem Boot-Service
(In99, Figure 2-1)

Vor der Ausführung kann der Client eine Überprüfung der Authentizität anfordern. Über eine DHCP Request Nachricht erhält dieser das Credentials File für die heruntergeladene Datei vom Boot Server und kann anschließend den Test durchführen.
Mit der Ausführung der Boot-Datei nach erfolgreichem Authentizitätstest und dem Hochfahren des Betriebssystems ist das Netzwerk-Booten über PXE abgeschlossen. (In99)

### 2.3.2 Trivial File Transfer Protocol

TFTP, das Kurzwort für Trivial File Transfer Protocol, ist ein einfach gestaltetes Dateiübertragungsprotokoll und wurde 1981 vom MIT-Informatiker und Internet-Pionier Noel Chiappa entwickelt (Ie81) und in Zusammenarbeit mit anderen Wissenschaftlern bis 1992 mit der Veröffentlichung der Request For Comments (RFC) Nummer 1350 modifiziert und verbessert.

Das Protokoll ist die vereinfachte Version des mit mehr Funktionen ausgestatteten File Transfer Protocol (FTP). Im Vergleich zu diesem kann es nur Dateien von einem Server lesen und schreiben, wodurch es allerdings weniger Speicherplatz benötigt und leicht implementierbar bleibt. Die heutige Hauptaufgabe liegt vor allem im Laden von Betriebssystemen und deren Konfigurationen über das Netzwerk.
TFTP gehört im Schichtenmodell zur Anwendungsschicht und ist oberhalb des verbindungslosen Transportprotokolls UDP (User Datagram Protocol) zu finden, welches die grundlegende Funktionalität der Datenübermittlung übernimmt.
Die Etablierung einer TFTP-Verbindung beginnt mit dem Senden einer Anfrage an Standard-Port 69, das Protokoll ermöglicht dabei in Verbindung mit dem Funktionsumfang eine Write-Request (WRQ) und eine Read-Request (RRQ). Das entsprechende Paket (Abb. 5) unterscheidet sich jeweils in der Zahl `opcode`. Eine Leseanfrage besitzt die Code-Nummer 1, die Schreibanfrage die Nummer 2.

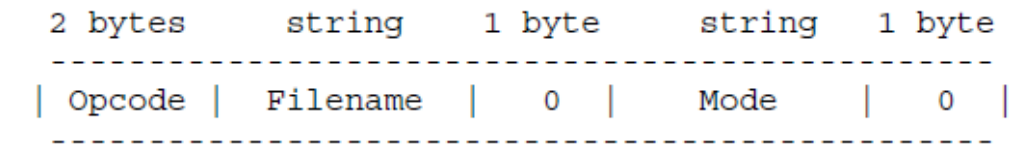


Abbildung 5: Aufbau TFTP-Request
((Ie92), Figure 5-1)

Nach zwei Byte `opcode` folgt der Name der zu übermittelten Datei als Klartext-String in `netASCII`-Code, einem ASCII mit Telnet-Modifikationen (8-Bit-ASCII). Dieser wird durch einen Null-Byte abgeschlossen. Nachfolgend befindet sich der Transfermodus ebenfalls als String in `netASCII`. TFTP kennt drei Modi, wobei der Modus `mail` schon 1992 als veraltet eingestuft wurde. Die übrigen sind wiederum `netASCII` und `octet`. Während `netASCII`, wie bereits beschrieben als 8-Bit-ASCII sendet, arbeitet die Methode `octet` mit einer einfachen Byte-Kodierung. Das Modus-Feld endet mit einer abschließenden Null, die gleichzeitig das Ende des Headers darstellt.
Sowohl Client als auch Server wählen jeweils unabhängig voneinander und nach dem Zufallsprinzip einen Transfer Identifier (TID), der im Laufe der Datenübermittlung als Quell- und Ziel-Port der jeweiligen Seite verwendet wird. Die Verbindung basiert also nicht auf festen Ports. In der Anfrage an Port 69 sendet der Client seine TID an den Server, dieser nutzt die Information für die darauffolgenden Pakete als Ziel-Port und übermittelt seine eigene TID als neue Nachrichtenquelle.

Je nachdem, ob eine Read- oder eine Write-Request gesendet wird, entstehen zwei unterschiedliche Protokollabläufe. Hat ein Client den Wunsch eine Datei zu schreiben, sendet er die entsprechende Anfrage an den Port 69. Der Server antwortet dabei entweder mit einem Acknowlegdement- (ACK) oder im Falle eines Fehlers mit einem Error-Paket. Geht man vom einem erfolgreichen Ablauf aus, wird ein ACK-Paket nach dem Aufbau der Abbildung 6 verschickt.

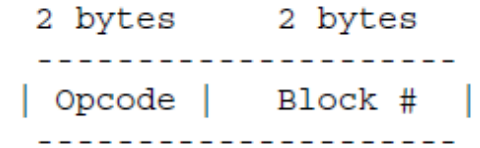


Abbildung 6: Aufbau Acknowlegdement-Paket
((Ie92), Figure 5-3)

Der Paket-Header wird mit der `opcode`-Nummer 4 versendet. Zusätzlich zu den zwei Byte Operation Code enthält die Nachricht die zu bestätigende Block-Zahl in weiteren zwei Byte. Auf eine Schreib-Anfrage antwortet der Server mit der Ziffer 0 und teilt damit seine Bereitschaft zum Empfang der Datei mit. Daraufhin kann der Client den ersten Datenblock senden. TFTP arbeitet mit konstanten Blöcken von 512 Byte, die in aufsteigender Reihenfolge beginnend mit eins nummeriert sind.
Ein Data-Paket ähnelt dem ACK-Paket, enthält aber zusätzlich den Datenblock, der mit diesem gesendet wird. Bei einer Größe von genau 512 Byte wird ein weiterer Block erwartet, ein Datenblock mit kleinerer Größe gilt als letztes Element und beendet die Übertragung. Das Datenpaket (Abb. 7) ist an der `opcode`-Zahl 3 erkennbar, hierbei erfolgt die eindeutige Zuordnung jedes Datenbereichs zu einer Blocknummer, die vor allem für die Bestätigung durch das Gegenüber benötigt wird.

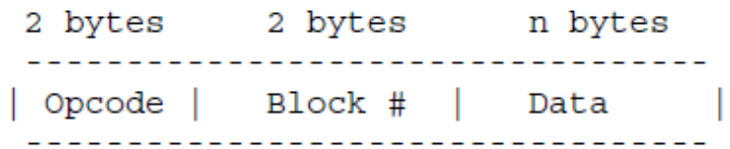


Abbildung 7: Aufbau Data-Paket
((Ie92), Figure 5-2)

Ist das Paket erfolgreich übermittelt worden, bestätigt der Server wiederum mit einem ACK-Paket, diesmal allerdings mit dem Eintrag für die

Blockzahl 1. Danach kann der Sendevorgang des zweiten Datenpaketes beginnen. Dieser Vorgang wiederholt sich, wie oben bereits beschrieben, bis ein Block mit einer Größe kleiner 512 Byte registriert wird. Dann bestätigt der Server ein letztes Mal mit einem finalen ACK und die Verbindung wird geschlossen.
Bei der zweiten Möglichkeit, eine Datei vom TFTP-Server zu lesen, ändert sich die Reihenfolge der vorkommenden Paketarten. Doch auch in diesem Fall sendet der Client zunächst eine Anfrage an Port 69 des Servers. Bis auf den `opcode` gleicht der Aufbau dem der Schreib-Anfrage (siehe Abb. 5).
Der Client erhält vom Server aber keine Bestätigung der Anfrage in Form eines ACK-Paketes, sondern bekommt sofort das erste Daten-Paket mit der Blocknummer 1 übersendet. Nun hat der Client die Aufgabe mit einer entsprechenden ACK-Nachricht den Erhalt dieses Datenblocks zu bestätigen. Nach Eingang des ACK 1 beim Server sendet dieser das zweite Datenpaket. Der Lese-Zyklus wiederholt sich, bis die angeforderte Datei vollständig übertragen wurde. Die Verbindung endet, wenn der Client final bestätigt. (Ie92)

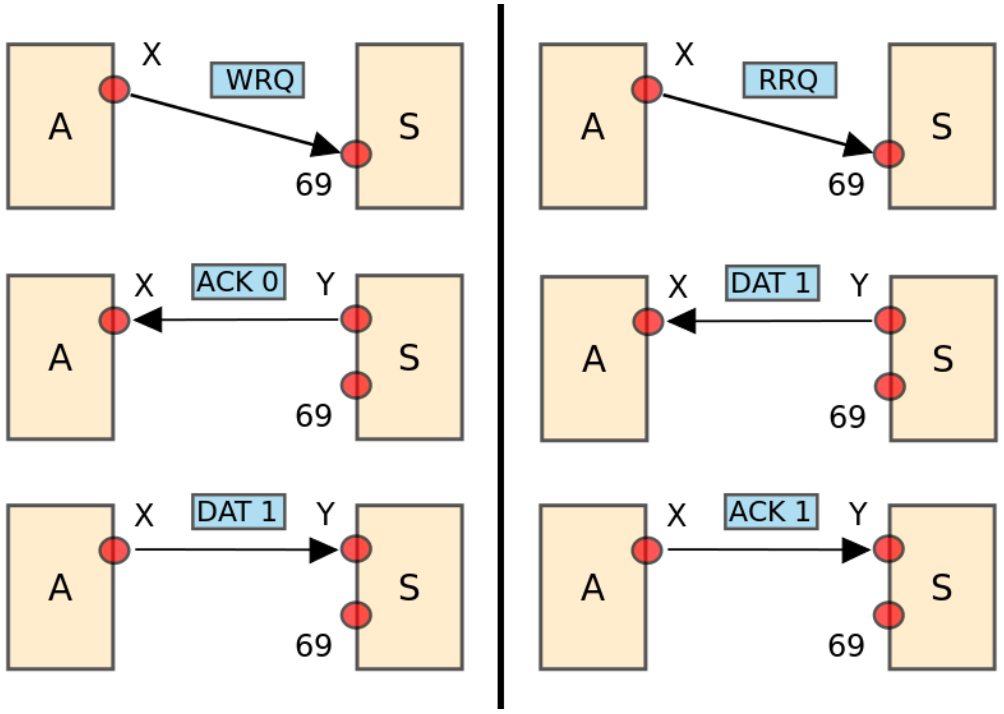


Abbildung 8: Zusammenfassung TFTP-Verbindungen
(https://en.wikipedia.org/wiki/Trivial_File_Transfer_Protocol)

Beide Aufgaben, zu denen der TFTP-Server in der Lage ist, sind in der Abbildung 8 noch einmal bildlich im Vergleich zu sehen. Dabei ist links der

Schreibvorgang und rechts der Ablauf beim Lesen vom TFTP-Server dargestellt. Die Variablen X und Y in der Abbildung stehen jeweils für die TID beziehungsweise die verwendete Port-Nummer des Clients A und des Servers S.
In dieser Arbeit ist vor allem die TFTP-Lesefunktion von Bedeutung, um in der Methodik das Laden des Kernels und der Device Tree Blobs (.dtb-Dateien) zu ermöglichen.

### 2.3.3 Network File System

Das Network File System, abgekürzt als NFS bezeichnet, gehört zu den Netzwerkprotokollen und existiert in den Versionen v2, v3 und v4. Es wurde ursprünglich 1989 von Sun Microsystems mit der RFC 1094 veröffentlicht und gilt als Internet-Standard.
NFS basiert auf dem Kommunikationsstandard External Data Representation (XDR), einem einheitlichen Schema zur Datendarstellung, und auf RPC, dem Remote Procedure Call, einer Prozedur zum Befehlsaufruf auf entfernten Plattformen. Während NFS im Schichtenmodell die Anwendungsschicht belegt, sind XDR in der darunter liegenden Darstellungs- und RPC in der wiederum eine Stufe niedrigeren Sitzungsschicht zu finden. Auf diese Weise lässt sich auch die Funktionszuteilung der einzelnen Bestandteile verstehen. Die beiden in NFS angewandten Techniken wurden ebenfalls von Sun entwickelt und ermöglichen durch ihre Plattform- und System-Unabhängigkeit den ebenfalls unabhängigen Einsatz von NFS.
Der Server des Network File Systems ist statuslos, das heißt er benötigt keine Informationen über den aktuellen Zustand seiner Clients, um fehlerfrei zu funktionieren. Bei einem Absturz von Client oder Server ist es dabei ohne Schwierigkeiten und Veränderungen möglich, einen von beiden oder beide neu zu starten und die vorgesehenen Aufgaben weiterzuführen.
Der Server stellt die implementierten Funktionalitäten zur Verfügung, ist darüber hinaus aber nicht fähig, weitere Aufgaben wie Debuggen oder Konvertierung vorzunehmen, diese Vorgänge muss der Client übernehmen.
Für eine NFS-Verbindung kann ein allgemeiner schematischer Ablauf definiert werden, der von den NFS-Versionen grundlegend befolgt wird und in den folgenden Absätzen dargelegt werden soll:
Möchte ein Client mit einem NFS-Server in Verbindung treten, benötigt er zunächst die IP-Adresse und den Port desselben. Da das Network File System ein Dienst von vielen weiteren innerhalb der Remote Procedure Call-Technik ist und daher keine festen Adressen und Ports existieren, müssen diese Verbindungsinformationen über den Portmapper beziehungsweise RPC-Bind über dessen Port 111 erfragt werden. Dieser liest die Werte

des angefragten Dienstes aus der ihm untergebenen Datenbank aus und gibt sie an den Client zurück; weitere Verbindungen finden daraufhin direkt mit dem nun bekannten Port des NFS-Servers statt. Dieser ist zur Durchführung der von ihm zur Verfügung gestellten Funktionen mit Deamons ausgestattet, die im Verlauf der Verbindungsprozedur einzeln angesprochen werden.
Der Client benötigt daraufhin den Port des Mount-Deamons, der es ermöglicht, die Inhalte im NFS-Server beim Client einzubinden. Dazu steht ihm erneut der Portmapper zu Diensten, im Normalfall erhält er von diesen den Port 694 zurück. Nun fähig den `mountd` zu kontaktieren, stellt der Client die Anfrage nach dem Filehandle für das von ihm gewünschte Verzeichnis. Der Deamon sendet ihm daraufhin ein Filehandle 0 für die Wurzel der einzubindenden Ordnerstruktur auf dem Server.
Der Client stellt erneut eine Anfrage an den Portmapper, um nun den Port des NFS-Deamons zu erfragen, über welchen er auf den Server zugreifen kann. Dieser antwortet mit der Port-Nummer 2049. Nach der Kontaktaufnahme mit dem NFS-Deamon kann der Client nun die verschiedenen Funktionen, die in NFS enthalten sind, aufrufen. (Ie95) (Ca00) Die folgenden Tabellen 5 und 6 zeigen eine Übersicht wichtiger Funktionalitäten eines NFSv3-Servers:

Tabelle 5: Zusammenfassung NFS-Server-Funktionen Teil 1

| Funktion | Hintergrund |
|---|---|
| GETATTR *fhdl* | Gibt die Attribute des mit dem Filehandle *fhdl* spezifizierten Dateisystemobjekts zurück. |
| SETATTR *fhdl attr* | Ändert ein Attribut *attr* des mit dem Filehandle *fhdl* spezifizierten Dateisystemobjekts. |
| LOOKUP *fhdl name* | Durchsucht das durch ein Filehandle *fhdl* definierte Verzeichnis nach einer Datei anhand des übergebenen Namens und gibt deren Attribute zurück. |
| ACCESS *fhdl acc* | Prüft die Zugriffsrechte eines Nutzers für die Datei mit dem Filehandle *fhdl*, die Berechtigungen werden als Bit-Maske *acc* übergeben. |
| READ *file off count* | Gibt eine Anzahl von *count* Bytes an Daten aus der mit *file* festgelegten Datei beginnend ab dem Startoffset *off* zurück. |
| WRITE *file off count* | Schreibt eine Menge von *count* Bytes an übergebenen Daten an eine mit *off* spezifizierte Stelle in der Datei *file*. |

(Ie95)

Tabelle 6: Zusammenfassung NFS-Server-Funktionen Teil 2

| Funktion | Hintergrund |
|---|---|
| CREATE *fhdl name mode* | Erstellt eine neue Datei *name* im Verzeichnis mit dem Filehandle *fhdl*. Der Modus *mode* ermöglicht die Prüfung von Dopplungen. |
| MKDIR *fhdl name attr* | Legt ein Unterverzeichnis *name* mit den zugehörigen Attributen *attr* im spezifizierten Ordner *fhdl* an. |
| REMOVE *fhdl name*<br>RMDIR *fhdl name* | Löscht ein Dateisystemobjekt (Datei, Link bzw. Unterordner) über dessen Eintrag *name* aus einem Verzeichnis *fhdl*. |
| RENAME *from to* | Benennt ein Objekt über dessen Eintrag oder ein Verzeichnis über den Filehandle (*from*) in den übergebenen Wert *to* um. |
| READDIR *fhdl* | Gibt die Anzahl an Einträgen im Verzeichnis zusammen jeweils mit Name und File Identifier zurück. Erweiterbar mit READDIRPLUS. |
| FSSTAT *fsroot* (dyn.)<br>FSINFO *fsroot* (stat.) | Ermöglicht das Abrufen von dynamischen oder statischen Dateisysteminformationen für das System unter dem Filehandle *fsroot*. |

(Ie95)

Der Client könnte nun beispielsweise mit dem Filehandle 0 des Mount-Deamons und dem Namen der gewünschten Datei die LOOKUP-Prozedur starten. Dabei wird das Server-Verzeichnis nach der angegebenen Datei durchsucht. Wenn sie existiert, erhält der Nutzer neben anderen Attributen das Filehandle 1 für das gefundene Dateisystemobjekt. Mit dieser Information stehen dem Client nun weitere Optionen zur Verfügung. Mit der READ-Funktion kann er sich etwa den Inhalt der Datei vom NFS-Deamon zusenden lassen. Auch das Schreiben in Dateien, die Erstellung und Löschung von Verzeichnissen oder die Anzeige beziehungsweise Anpassung von Attributen werden vom NFS-Server unterstützt.
Zusätzlich zur beschriebenen reinen NFS-Funktion nutzt das Protokoll die MOUNT-Prozedur. Diese wird benötigt, um den Server-Pfaden ein Filehandle zuordnen zu können, sodass ein Client den betreffenden Pfad einbinden kann. Weiterhin ist ein Network Lock Manager nötig, um dem statuslosen NFS unter anderem das Sperren von eingebundenen Ordnern zu ermöglichen. (Ie95)

# 3 Methode

Die nachfolgende Anleitung weist den Weg zum erfolgreichen Netzwerk-Boot eines Android-Abbildes auf dem Wandboard. Diese basiert mit einigen Änderungen auf bereits existierenden Anleitungen für Android 6.0 der Quellen (NC16) und (NR16). Für die Durchführung ist zunächst die Generierung der Android-Bootdateien als vorbereitender Schritt notwendig.

## 3.1 Vorbereitung / Bau des Betriebssystems

Zuerst müssen die Quelldateien des gewünschten Android-Systems für das Wandboard heruntergeladen werden. Diese sind als Archivdateien im tar.xz-Format im FTP-Download-Bereich der Webseite des Wandboard-Entwicklers (download.wandboard.org/) zu finden.
Nach dem Herunterladen des Sourcecodes sind im nächsten Schritt bestimmte Voraussetzungen für einen erfolgreichen System-Build zu prüfen. Zuallererst benötigt man als Host entweder ein Ubuntu- oder Mac OS-System. Diese Anleitung wurde an Linux Mint 17.3 Rosa erprobt, das auf Ubuntu basiert. Laut der Webseite des Android Open Source Project (AOSP) sind Erfahrungen mit anderen Linux-Distributionen nicht dokumentiert, diese sollen die benötigten Grundvoraussetzungen aber ebenso beinhalten. Windows wird nicht unterstützt. (An17)
Weiterhin ist das Vorhandensein der OpenJDK Version 7 notwendig. Andere Versionen oder die Oracle JDK sorgen für Fehler beim Kompilieren des AOSP. Ob die richtige JDK wie in Abbildung 9 vorhanden ist, kann in einer Konsole über den Befehl `java -version` geprüft werden.

```
clenk@christopherlenk-Satellite-L650 ~ $ java -version
java version "1.7.0_131"
OpenJDK Runtime Environment (IcedTea 2.6.9) (7u131-2.6.9-0ubuntu0.14.04.2)
OpenJDK 64-Bit Server VM (build 24.131-b00, mixed mode)
```

Abbildung 9: Beispiel des Vorhandenseins der korrekten JDK

Ist das passende Paket noch nicht installiert, kann das nach der Aktualisierung der Paketlisten (`sudo apt-get update`) mit dem Befehl `sudo apt-get install openjdk-7-jdk` erfolgen.
Zusätzlich muss beachtet werden, dass das Android Build-System die Java-Laufzeitumgebung über die Variable `JAVA_HOME` aufruft. Verweist diese auf eine andere Java-Version, ist gar nicht oder falsch definiert, kommt es im Laufe des make-Prozesses zu Fehlern und daraus resultierenden Abbrüchen. Die Variable kann mit `echo $JAVA_HOME` angezeigt und über `export JAVA_HOME` für das aktuelle Terminalfenster geändert werden (Abb. 10).

```
clenk@christopherlenk-Satellite-L650 ~ $ echo $JAVA_HOME
/usr/java/bin/java
clenk@christopherlenk-Satellite-L650 ~ $ export JAVA_HOME=/usr/lib/jvm/java-7-op
enjdk-amd64
clenk@christopherlenk-Satellite-L650 ~ $ echo $JAVA_HOME
/usr/lib/jvm/java-7-openjdk-amd64
```

Abbildung 10: Überprüfung und Änderung der Variable `JAVA_HOME`

Bevor der Bau beginnen kann, müssen einige benötigte Pakete eingerichtet werden, falls diese noch nicht vorhanden sind. Die betreffende Software ist im nachfolgend dargestellten Befehl aufgelistet:

```
sudo apt-get install git-core gnupg flex bison gperf lzop
build-essential zip curl zlib1g-dev gcc-multilib g++-multilib
libc6-dev-i386 lib32ncurses5-dev x11proto-core-dev libx11-dev
lib32z-dev ccache libgl1-mesa-dev libxml2-utils xsltproc unzip
u-boot-tools
```

Um das Kompilieren fehlerfrei zu ermöglichen und das System für die spätere Nutzung optimal einzurichten, ist es notwendig, am Inhalt einzelner Dateien Änderungen vorzunehmen.
Dazu passt man zunächst mit einem Editor die Quelldatei `HOST_x86_common.mk`, die unter `build/core/clang/` zu finden ist, wie in Abbildung 11 dargestellt, an:

```
CLANG_CONFIG_x86_DARWIN_HOST_EXTRA_CFLAGS := \
  -integrated-as

CLANG_CONFIG_x86_DARWIN_HOST_EXTRA_CFLAGS += -fstack-protector-strong
endif

ifeq ($(HOST_OS),linux)
CLANG_CONFIG_x86_LINUX_HOST_EXTRA_ASFLAGS := \
  --gcc-toolchain=$($(clang_2nd_arch_prefix)HOST_TOOLCHAIN_FOR_CLANG) \
  --sysroot $($(clang_2nd_arch_prefix)HOST_TOOLCHAIN_FOR_CLANG)/sysroot \
  -B$($(clang_2nd_arch_prefix)HOST_TOOLCHAIN_FOR_CLANG)/x86_64-linux/bin

CLANG_CONFIG_x86_LINUX_HOST_EXTRA_CFLAGS := \
```

Abbildung 11: Veränderungen in der Datei `HOST_x86_common.mk`

Nachfolgend muss die Standard-Konfiguration des Android-Frameworks geändert werden, um zu verhindern, dass das Wandboard bei Aktivierung des Ruhezustands durch Android einen Neustart durchführt. Dafür sind die in Abbildung 12 zu sehenden letzten zwei Zeilen über den bevorzugten Editor in die Datei `device/fsl/wandboard/overlay/frameworks/base/packages/SettingsProvider/res/values/defaults.xml` einzufügen:

```
<resources>
    <!-- Default for Settings.Secure.USER_SETUP_COMPLETE -->
    <bool name="def_user_setup_complete">true</bool>
    <!-- Disable GPS Provider-->
    <string name="def_location_providers_allowed" translatable="false"></string>
    <bool name="def_stay_on_while_plugged_in">true</bool>
    <integer name="def_screen_off_timeout">30000</integer>
</resources>
```

Abbildung 12: Erweiterung der Einstellungsdatei

Mit den gemachten Änderungen steht einem erfolgreichen Build-Prozess nichts mehr im Wege. Bevor das eigentliche Betriebssystem, das AOSP, kompiliert werden kann, müssen zuvor der Kernel und der Bootloader konfiguriert und gebaut werden, da das Build-System diese im einsetzbaren Zustand voraussetzt und einige Abhängigkeiten verknüpfen muss.
Zuvor ist es notwendig, für alle Ordner des Android-Sourcecodes die Build-Variablen `ARCH` und `CROSS_COMPILE` festzulegen. Dazu dient in Abbildung 13 erneut der Befehl `export`. Bei den beiden Werten handelt es sich um die zu erzeugende Prozessor-Architektur ARM und den Pfad für den Cross-Compiler, der im späteren Verlauf die Übersetzung der Android-Quelldateien auf dem Linux-Host übernehmen wird.

```
clenk@christopherlenk-Satellite-L650 ~/Schreibtisch/wandboard_android $ export A
RCH=arm
clenk@christopherlenk-Satellite-L650 ~/Schreibtisch/wandboard_android $ export C
ROSS_COMPILE="/home/clenk/Schreibtisch/wandboard_android/prebuilts/gcc/linux-x86
/arm/arm-eabi-4.6/bin/arm-eabi-"
```

Abbildung 13: Festlegung der Build-Variablen

Nun kann im ersten Abschnitt des Bauprozesses der Kernel konstruiert werden. Dazu wechselt man über `cd` in das Verzeichnis `kernel_imx`, in welchem die Systemkern-Dateien abgelegt sind. Zunächst muss die Verarbeitung der Konfiguration des späteren Kernels vorbereitet werden, indem der Befehl `make wandboard_android_defconfig` zum Einsatz kommt. Dieser verarbeitet Quelldateien, die die Konfigurationsgrundlagen enthalten. Entsprechend der dortigen Vorgaben werden die config-Dateien angelegt.
Nachfolgend ist es möglich, die Konfiguration über `make menuconfig` in einer Listenansicht zu verändern, die in Abbildung 14 zu sehen ist. Hierbei können beispielsweise Boot-Optionen gewählt, Treiber aktiviert und grundlegende Einstellungen wie Energieverwaltung oder Bus-Unterstützung angepasst werden. Die hier getroffenen Entscheidungen speichert das System in den config-Dateien ab, sodass alle Veränderungen im aktiven Kernel wieder zu finden sind.

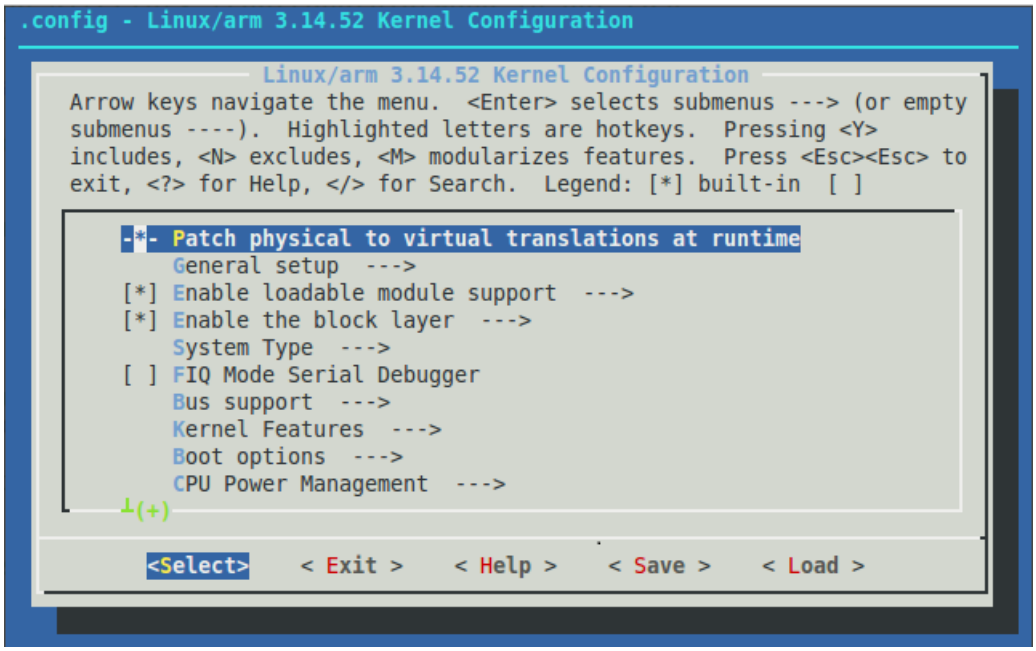


Abbildung 14: Konfiguration des Kernels über `menuconfig`

Mit Abschluss der Konfigurierung kann der Bau des Kerns beginnen. Dieser startet mit der Anweisung `make -j4 zImage`, wobei `zImage` als Name des späteren Kernel-Images übergeben wird. Der Parameter `-j4` schreibt dem make-Prozess vor, wie viele parallel laufende Kompilier-Vorgänge es geben soll. In diesem Fall werden also vier Abhängigkeiten aus dem Makefile gleichzeitig verarbeitet.
Der Vorgang benötigt je nach Systemvoraussetzungen einige Minuten Zeit. Dabei übersetzt der Compiler die C-Quelldateien, die anschließend in der vom Makefile vorgeschriebenen Anordnung mit Beachtung der Verbindungen und Abhängigkeiten zu einem System-Image zusammengeführt werden. Der Prozess ist erfolgreich abgeschlossen, wenn dieser mit den in Abbildung 15 dargestellten Meldungen beendet wird.

```
AS      arch/arm/boot/compressed/ashldi3.o
AS      arch/arm/boot/compressed/bswapsdi2.o
AS      arch/arm/boot/compressed/piggy.lzo.o
LD      arch/arm/boot/compressed/vmlinux
OBJCOPY arch/arm/boot/zImage
Kernel: arch/arm/boot/zImage is ready
```

Abbildung 15: Abschluss der Kernel-Kompilierung

Nunmehr ist es nötig, den Bootloader herzustellen, der das Laden des Betriebssystems beziehungsweise den Netzwerk-Boot übernehmen soll. Dessen Quelldateien sind im heruntergeladenen Android-Archiv bereits enthalten und über den Pfad `bootable/bootloader/uboot-imx` erreichbar. Bevor begonnen wird, sollte die korrekte Definition der Variablen `ARCH` und `CROSS_COMPILE` nachgeprüft werden, die auch für die Übersetzung von U-Boot essenziell sind.
Befindet man sich im oben genannten Verzeichnis, muss zunächst mit der Anweisung `make wandboard_defconfig` die Konfiguration geladen und abgespeichert werden. Danach ist es möglich, auf eine Art und Weise ähnlich der Kernel-Konstruktion mithilfe des leicht abgeänderten Befehls `make -j4` den Bau des Bootloaders zu starten. Das U-Boot-Image ist einsatzbereit, wenn der Prozess mit der in Abbildung 16 dargestellten Ausgabe endet. Die Kompilierung dauert je nach Host zwischen einigen Sekunden und ein bis zwei Minuten.

```
OBJCOPY u-boot.srec
OBJCOPY u-boot.bin
CC      spl/common/cmd_disk.o
MKIMAGE u-boot.img
LD      spl/lib/built-in.o
LD      spl/common/built-in.o
LD      spl/u-boot-spl
OBJCOPY spl/u-boot-spl.bin
CFGS    arch/arm/imx-common/spl_sd.cfg.cfgtmp
MKIMAGE SPL
```

Abbildung 16: Erfolgreicher Abschluss der Bootloader-Erstellung

Mit dem erfolgreichen Abschluss der Kernel- und U-Boot-Konstruktion sind alle Voraussetzungen erfüllt, um das Betriebssystem zu übersetzen und dessen Bibliotheken und Dateien zusammenzustellen. Hierzu ist notwendig, über `cd` in den obersten Ordner des ehemaligen Android-Quellarchivs zu wechseln, damit alle Sourcecode-Dateien zugreifbar sind. In diesem Ordner wird im folgenden Schritt die Kompilierungsumgebung initiiert und ausgeführt. Der Befehl `. build/envsetup.sh` ruft das bezeichnete Shell-Skript auf, welches wiederum weitere Skripte einbindet, die alle verfügbaren Gerätetypen und Prozessorarchitekturen, für die Android eingerichtet werden kann, lädt und diese für die Kompilierung vorbereitet.
Das aufgerufene Skript vollführt die Vorbereitung automatisch und ermöglicht die darauffolgende Anweisung `lunch`. Mit dieser erhält der Nutzer eine Auflistung verschiedener Produkte, für deren Architektur der Build-Prozess möglich ist. In diesem Fall muss in Abbildung 17 der Eintrag mit der Nummer 23 ausgewählt werden, der für die Spezifikation `wandboard-eng` steht.

```
     22. evk_6ul-user
     23. wandboard-eng
     24. wandboard-user
     25. sabresd_7d-eng
     26. sabresd_7d-user
     27. mini_emulator_arm64-userdebug
     28. m_e_arm-userdebug
     29. mini_emulator_mips-userdebug
     30. mini_emulator_x86_64-userdebug
     31. mini_emulator_x86-userdebug
     32. aosp_flounder-userdebug
     33. aosp_angler-userdebug
     34. aosp_bullhead-userdebug
     35. aosp_hammerhead-userdebug
     36. aosp_hammerhead_fp-userdebug
     37. aosp_shamu-userdebug

Which would you like? [aosp_arm-eng] 
```

Abbildung 17: Auswahl Produktarchitektur

Für die gewählte Eintragung erstellt `lunch` eine Konfiguration für das spätere Android, die der Nutzer über `printconfig` (Abb. 18) auf Richtigkeit überprüfen kann.

```
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=6.0.1
TARGET_PRODUCT=wandboard
TARGET_BUILD_VARIANT=eng
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a-neon
TARGET_CPU_VARIANT=cortex-a9
TARGET_2ND_ARCH=
TARGET_2ND_ARCH_VARIANT=
TARGET_2ND_CPU_VARIANT=
HOST_ARCH=x86_64
HOST_OS=linux
HOST_OS_EXTRA=Linux-3.19.0-32-generic-x86_64-with-LinuxMint-17.3-rosa
HOST_BUILD_TYPE=release
BUILD_ID=1.0.0-ga-rc6
OUT_DIR=out
```

Abbildung 18: Ausgabe des Befehls `printconfig`

Bevor der eigentliche Bau des Systems beginnen kann, ist es notwendig, den Inhalt der Variable `JAVA_HOME` nochmals zu überprüfen (siehe Abb. 10), da die Laufzeitumgebung des Build-Prozesses alle Werkzeuge, also auch den dort angegebenen Pfad, auf Richtigkeit und Java-Version in Bezug auf die Kompilierung kontrolliert.

Über den Befehl `make -j8` kann der make-Prozess gestartet werden, der die Übersetzung des AOSP durchführt. Dabei wird die zuvor erstellte Konfiguration übernommen, die Werkzeugprüfung erfolgt und alle benötigten Ordner und Dateien werden nacheinander eingebunden. Dieser Vorgang kann einige Stunden in Anspruch nehmen und endet mit dem Erzeugen des Dateisystems, das heißt mit der Bereitstellung des RAM-Disk-Images und des Abbilds der System-Partition.
Die Ausgaben, die das Endergebnis symbolisieren und einen erfolgreichen Abschluss darstellen, sind in Abbildung 19 zu sehen.

```
Creating filesystem with parameters:
    Size: 624152576
    Block size: 4096
    Blocks per group: 32768
    Inodes per group: 7632
    Inode size: 256
    Journal blocks: 2380
    Label: system
    Blocks: 152381
    Block groups: 5
    Reserved block group size: 39
Created filesystem with 1530/38160 inodes and 93182/152381 blocks

Install system fs image: out/target/product/wandboard/system.img
out/target/product/wandboard/system.img+out/target/product/wandboard/obj/PACKAGING/
recovery_patch_intermediates/recovery_from_boot.p maxsize=642318336 blocksize=4224
total=378917909 reserve=6488064
simg2img out/target/product/wandboard/system.img out/target/product/wandboard/syste
m_raw.img;
```

Abbildung 19: Beendung der Dateisystemerstellung (gekürzt)

Diese für die Durchführung des Netzwerkboots wichtigen Dateien sind nach Abschluss des Build-Prozesses unter dem Pfad `/out/target/product/wandboard/` sowie im Kernel- und im U-Boot-Verzeichnis auffindbar.

## 3.2 Umwandlung der Abbilder

Um die benötigten Dateien für die Durchführung des Boots von Android-Images zu erhalten, kann neben dem Kompilierungsansatz auch die Überlegung verfolgt werden, die Dateien aus den SD-Karten-Images herauszulösen und umzuwandeln. Die Umwandlung ist nötig, weil die dort zu findenden Inhalte mit dem Netzwerkstart nicht direkt kompatibel sind. In den folgenden Abschnitten wird dabei die alternative Generierung jeder benötigten Datei einzeln betrachtet.

**zImage und .dtb-Datei** Sowohl die Kernel-Datei `zImage` als auch die Binärdatei des Device Tree Blobs liegen auf der SD-Karte bereits in der Form vor, wie sie für den Bootvorgang benötigt werden, sodass es möglich ist, diese ohne zusätzliche Umwandlung in das Verzeichnis des TFTP-Servers zu überführen.

**system_raw.img** Auf der SD-Karte liegt kein System-Image vor, dessen Dateien sind als Ordnerstruktur in der entsprechenden Partition eingebunden. Über den Befehl `dd` kann diese in ein Image eingelesen werden. Dabei entsteht allerdings kein Abbild, das zum Mounten fähig ist. Diese Aufgabe übernimmt der beim Kompilieren für die Erstellung des Rohabbilds angewandte Befehl `simg2img system.img system_raw.img`, dieser kann jedoch auch alleinstehend außerhalb des Build-Prozesses verwendet werden.

**ramdisk.img** Das RAM-Disk-Image ist im SD-Karten-Abbild enthalten, allerdings in der Datei `uramdisk.img`. Diese besitzt einen 64 Byte großen U-Boot-Header und die Inhalte der RAM-Disk. Um diese Daten zu extrahieren, kann das Tool `dd` eingesetzt werden. Dabei muss der Befehl so gestaltet sein, dass die ersten 64 Byte, also der nicht benötigte Header, nicht verarbeitet werden. Eine solche Anweisung ist im folgenden dargestellt:
`sudo dd if=uramdisk.img of=ramdisk.gz bs=64 skip=1`
Der Prozess erzeugt ein .gz-Archiv, das alle erforderlichen Dateien und Strukturen enthält. Mithilfe von `mv ramdisk.gz ramdisk_new.img` kann das Archiv in das gewünschte Image-Format für den NFS-Server gebracht werden.

## 3.3 Konfiguration des Netzwerkstarts

Die im vorherigen Schritt fertiggestellten Teile des Android-Systems für das Wandboard sollen im folgenden nicht wie normalerweise auf eine SD-Karte gebracht werden, um starten zu können. Mithilfe des Bootloaders U-Boot und der Protokolle TFTP und NFS ist es möglich, Android über das Netzwerk zu booten.
Zunächst müssen die Server der beteiligten Protokolle auf dem Linux-Host installiert und konfiguriert werden. Für den TFTP-Server ist, wenn noch nicht auf dem System vorhanden, die Installation des Paketes `tftpd-hpa` nötig, die über den bereits mehrmals genutzten Befehl `sudo apt-get install` erfolgt. Mit dem erfolgreichen Abschluss der Einrichtung des TFTP-Deamons kann nun die Konfiguration des späteren Servers vorgenommen werden. Die betreffende Config-Datei `/etc/default/tftpd-hpa` enthält bereits die benötigten

Informationen für einen erfolgreichen Transfer. In diesem Fall (Abb. 20) wurde ausschließlich die Pfadangabe zum TFTP-Verzeichnis, in welchem die zu übertragenden Daten liegen, angepasst.

```
# /etc/default/tftpd-hpa

TFTP_USERNAME="tftp"
TFTP_DIRECTORY="/srv/tftpboot"
TFTP_ADDRESS="0.0.0.0:69"
TFTP_OPTIONS="--secure --create"
```

Abbildung 20: Inhalt der TFTP-Konfigurationsdatei

Das soeben angegebene Verzeichnis `tftpboot` muss mithilfe von `sudo mkdir /srv/tftpboot` erstellt und im folgenden dem TFTP-User `tftp` als Eigentümer zugeschrieben werden. Dafür dient die Anweisung `sudo chown -R tftp /srv/tftpboot`, der Parameter `-R` schließt rekursiv alle möglichen Unterverzeichnisse und Dateien mit ein (Abb. 21).

```
clenk@christopherlenk-Satellite-L650 /srv $ ls -l
insgesamt 8
drwxr-xr-x 16 root root 4096 Jun 23 10:00 nfsandroid
drwxr-xr-x  2 tftp root 4096 Jun 15 12:07 tftpboot
```

Abbildung 21: Besitzer und Rechte des TFTP- und NFS-Ordners

Der TFTP-Service ist nach Abschluss dieser Vorgänge startbereit und kann mit dem Befehl `sudo service tftpd-hpa start` aktiviert werden. Alternativ ist es möglich, diesen mit `stop` anzuhalten, mit `restart` neu zu starten und mit `status` den aktuellen Zustand abzurufen. Der TFTP-Server ist beim Netzwerkstart für die Bereitstellung des Kernels und der Device-Tree-Files zuständig. Die entsprechenden Dateien müssen, wie in Abbildung 22 zu sehen, im Verzeichnis `tftpboot` platziert werden, um bei der Durchführung des Bootens zur Verfügung zu stehen.

```
clenk@christopherlenk-Satellite-L650 /srv/tftpboot $ ls
imx6dl-wandboard.dtb  imx6q-wandboard.dtb  zImage
```

Abbildung 22: Ansicht des Ordners `tftpboot`

Um nun auch einen NFS-Server aufsetzen zu können, ist zunächst die Installation des zugehörigen Paketes mit dessen Abhängigkeiten vorzunehmen. Ähnlich dem TFTP-Server wird diese mit der Anweisung `sudo apt-get install nfs-kernel-server` durchgeführt, falls auf dem Host noch kein NFS-Server vorhanden sein sollte.

Die wichtigste Konfigurationsdatei ist die Textdatei `/etc/exports`, die den Zugriff von NFS-Clients auf ein Verzeichnis mit einem Dateisystem beziehungsweise auf mehrere Verzeichnisse mit unterschiedlichen Systemen regelt. Die Zugriffskontrolle erfolgt über die IP-Adresse oder den Hostnamen, dieser muss allerdings mit passender IP in der Datei `/etc/hosts` eingetragen sein. Einem NFS-Ordner können dabei mehrere Client-Adressen mit verschiedenen Optionen zugeordnet werden. Die folgende Abbildung 23 zeigt den `/etc/exports`-Eintrag für den in diesem Fall verwendeten Wurzel-Pfad `/srv/nfsandroid`, der durch die Nutzung eines Editor wie beispielsweise `nano` eingefügt werden kann, dabei müssen allerdings root-Rechte mithilfe des Befehls `sudo` vorliegen.

```
# /etc/exports: the access control list for filesystems which may be exported
#               to NFS clients.  See exports(5).
#
# Example for NFSv2 and NFSv3:
# /srv/homes       hostname1(rw,sync,no_subtree_check) hostname2(ro,sync,no_subtr$
#
# Example for NFSv4:
# /srv/nfs4        gss/krb5i(rw,sync,fsid=0,crossmnt,no_subtree_check)
# /srv/nfs4/homes  gss/krb5i(rw,sync,no_subtree_check)
#

/srv/nfsandroid 192.168.2.102/24(rw,no_root_squash,sync,crossmnt,no_subtree_check)
```

Abbildung 23: Eintragung in der Konfigurationsdatei `/etc/exports`

Der in der Konfiguration spezifizierte Ordner muss durch `sudo mkdir /srv/nfsandroid` erstellt werden. Nun ist es möglich, auf Grundlage der Exportdatei mit dem Befehl `sudo exportfs -a` die NFS-Tabelle zu bilden. Mit Abschluss dieser Schritte kann der Start des NFS-Servers erfolgen; dazu dient die Anweisung `sudo service nfs-kernel-server start`, die Alternativen `stop`, `restart` und `status` gelten wie beim TFTP-Service auch hier. Nun gilt es den NFS-Server mit dem Android-Betriebssystem zu versorgen, welches auf dem Wandboard gestartet werden soll. Hierbei kommen die nach einer der beiden Ansätze erhaltenen Abbilder der RAM-Disk und der System-Partition zum Einsatz.
Zunächst muss in den soeben erstellten Ordner des NFS-Servers gewechselt werden. In diesen ist die Image-Datei `ramdisk.img` zu entpacken, die wiederum ein gz-Archiv enthält. Der dadurch angewendete Befehl mit Pipe
`gunzip -c [Pfad zum Quellverzeichnis]/out/target/product/wandboard/ramdisk.img | sudo cpio -i`
überträgt den gesamten Inhalt mitsamt des Wurzelverzeichnisses aus dem Abbild in den Server.
In der ausgepackten Ordnerstruktur ist bereits ein Ordner `system` enthalten,

der zu Beginn allerdings noch leer ist. In diesen muss das `system_raw.img` eingebunden werden. Mithilfe von

```
sudo mount [Pfad zum Quellverzeichnis]/out/target/product/
wandboard/system_raw.img system
```

wird der Mount erfolgreich vollzogen. Damit enthält der NFS-Server nun sowohl RAM-Disk als auch System-Partition, sodass beides in einem Schritt bootfähig ist.
Die Server sind nunmehr mit allen Dateien ausgestattet, die für einen erfolgreichen Netzwerkstart benötigt werden. Im nächsten Schritt erfolgt der Start und die Konfiguration des Bootloaders, der nach Abschluss dieser mit den eingerichteten Servern kommunizieren, Daten empfangen und verarbeiten soll, sodass Android hochfährt, ohne auf die SD-Karte zuzugreifen.
Die SD wird aber trotz dieser Funktion noch benötigt, um den Bootloader bereitzustellen. Daher ist es nötig, eine SD-Karte mit den U-Boot-Dateien aus dem Build-Prozess zu bestücken. Die Binärdatei des Secondary Program Loaders (SPL), der das U-Boot in den Arbeitsspeicher lädt, muss an Offset 1k (1024) platziert werden. Dazu kann das Kopier-Tool `dd` Anwendung finden, zuvor sollte man beispielsweise mit dem Kommando `lsblk` die Kennung der Speicherkarte unter den Gerätedateien ermitteln. Dieser Gerätepfad und vor allem dessen letzter Buchstabe ersetzt im nachfolgenden Befehl den Platzhalter `/dev/sd#`:

```
sudo dd if=[Pfad zum Quellverzeichnis]/bootable/bootloader/
uboot-imx/SPL of=/dev/sd# bs=1k seek=1
```

Der Befehl `sync` schreibt die Daten aus dem Speicher auf die oben spezifizierte Gerätedatei, die stellvertretend für die SD-Karte existiert.
Neben dem SPL ist auch das Schreiben des U-Boot-Images an die richtige Position essentiell. Das Abbild muss in die erste FAT32-Partition, in der Partitionierung zumeist als `boot` bezeichnet, eingefügt werden, da der SPL dieses dort sucht und aufruft. Das beschriebene Vorhaben ist über die nachfolgend gelistete Anweisung durchführbar:

```
cp [Pfad zum Quellverzeichnis]/bootable/bootloader/uboot-imx/
u-boot.img /media/$USER/boot
```

Alternativ lässt sich auch das vorinstallierte U-Boot eines vollständig konfigurierten Android-SD-Abbildes verwenden.
Das Wandboard ist nun entweder mit einem kompilierten oder einem vorinstallierten Bootloader ausgestattet, an dem im folgenden Abschnitt einige Konfigurationen vorgenommen werden, um vorerst einen manuellen und im späteren Verlauf den automatischen Netzwerkboot zu ermöglichen.
Um mit dem U-Boot kommunizieren zu können, wird eine Verbindung zwischen der seriellen Schnittstelle des Wandboards und dem Host über ein Nullmodemkabel benötigt. Besitzt der Server oder PC keinen eigenen seriel-

len Anschluss, ist ein USB-zu-Seriell-Adapter notwendig. Auf dem Host kann der Datenverkehr über ein entsprechendes Konsolen-Tool für serielle Verbindungen wie `kermit`, `screen` oder `putty` empfangen werden.
Wird das Wandboard an die Energieversorgung angeschlossen, fährt nach dem SPL und dessen Aufruf der Bootloader hoch und listet einige Informationen über die vorliegende Hardware auf, bevor er einen fünf Sekunden dauernden Countdown zum normalen Booten startet. Verhindert man das Ablaufen der Zeitschranke durch Drücken einer beliebigen Taste, erhält man Zugang zur U-Boot-Konsole (Abb. 24), die eine vielfältige Befehlslandschaft zur Anpassung des Bootloaders bietet.

```
U-Boot 2015.04 (Jun 02 2017 - 15:04:21)

CPU:   Freescale i.MX6Q rev1.5 at 792 MHz
CPU:   Temperature 23 C
Reset cause: POR
Board: Wandboard
I2C:   ready
DRAM:  2 GiB
MMC:   FSL_SDHC: 0, FSL_SDHC: 1
auto-detected panel HDMI
Display: HDMI (1024x768)
In:    serial
Out:   serial
Err:   serial
Net:   FEC [PRIME]
Normal Boot
Hit any key to stop autoboot:  0
=>
```

Abbildung 24: Start des Bootloaders

Der Netzwerkstart ist im U-Boot bereits in den Umgebungsvariablen `netboot` und `netargs` definiert, sodass nur die Netzwerk- und Datei-Variablen an die Werte von Server und Client angepasst werden müssen. Für alle weiteren Schritte ist es nunmehr zwingend erforderlich, über den Ethernet-Anschluss des Wandboards eine Verbindung zum LAN herzustellen, der Host benötigt ebenfalls Zugang zum gleichen Netzwerk und Subnetz.
Dem Bootloader muss zunächst eine IP-Adresse im lokalen Netz zugewiesen werden. Bevor der Aufruf des DHCP-Clients erfolgen kann, bedarf es der Festlegung der Variable `autoload`, die mithilfe von `setenv autoload no` auf Nein gesetzt wird, um die automatische Funktion des Kernel-Ladens über DHCP abzuschalten. Der Client, dessen Start über die Anweisung `dhcp` erfolgt, übermittelt dem U-Boot daraufhin die aktuelle IP-Adresse im angeschlossenen Netzwerk. Diese ist als Variable `ipaddr` festzuhalten, wozu der Befehl `setenv ipaddr [IP-Adresse]` angewendet wird.

Auch die IP des Servers muss dem Bootloader in Form der Variable `serverip` mitgeteilt werden. Mit Ausnahme von Variablenname und IP-Adresswert gleicht der Befehl in Form und Anordnung der oberen Anweisung zur Registrierung der U-Boot-IP.
Der Variablenwert von `ip_dyn` ist auf gleiche Art und Weise wie beim Wert `autoload` mit `setenv` auf Nein zu setzen, um TFTP als Ladeprotokoll für Kernel und Device Tree Blob zuzulassen. Deren Dateinamen müssen für den erfolgreichen Ablauf des TFTP-Ladens wie folgt über die Anweisungen `setenv bootfile zImage` und `setenv fdtfile imx6q-wandboard.dtb` registriert werden. Weiterhin besteht die Notwendigkeit, den Namen des NFS-Wurzelverzeichnisses über `setenv nfsroot /srv/nfsandroid` in die Konfiguration einzutragen.
Abschließend ist es nötig, die bereits existierende Variable `netargs` durch die nachfolgend dargestellte angepasste Befehlskette zu modifizieren:

```
setenv netargs 'setenv bootargs $bootargs_base root=/dev/nfs rw
ip=dhcp nfsroot=$serverip:$nfsroot,v3,tcp'
```

Dabei erfolgt die Änderung der Argumente für den Netzwerkstart, indem die Boot-Parameter durch die Hinzufügung der Basis-Konfiguration (Videoeinstellungen, Lage der init-Datei, Werte der Terminal-Konsole, etc.) erweitert werden. Zusätzlich erfolgt die Festlegung des Root-Device auf die stellvertretende Gerätedatei für NFS, der Inhalt dieser wird dabei mit Lese- und Schreibrechten (rw) eingebunden. Die IP-Konfiguration besonders in Bezug auf die Adressierung soll über DHCP stattfinden, die Angabe des NFS-Wurzelverzeichnisses wird durch die Voranstellung der IP-Adresse des Servers ergänzt und die gesamte Verbindung zu diesem wird auf Grundlage von NFS Version 3 und TCP etabliert werden.
Alle getätigten Festlegungen sind nun final zu sichern, indem `env save` aufgerufen wird, das alle Variablenänderungen in den Speicher des Bootloaders schreibt. Mit `env default -a` können beispielsweise im Fall eines Fehlers wiederum alle Änderungen verworfen und U-Boot auf Standard zurückgesetzt werden.
Die Funktion des Netzwerkstarts kann manuell über den Aufruf `run netboot` getestet werden. Dabei sollte ein teilweise erfolgreicher, aber nicht vollständig ablaufender Android-Boot zu sehen sein. Um diese Schwierigkeiten zu beseitigen, sind in den abschließenden Schritten einige Änderungen in System- und Quelldateien nötig.
Zuerst muss dafür die Datei `fstab.freescale` angepasst werden, die die Verwaltung der Partitionen des Dateisystems zur Aufgabe hat. Da die System-Partition schon im NFS-Server in die RAM-Disk eingebunden ist, wäre es überflüssig, sie als Partition der SD-Karte zu mounten. Den entsprechenden Eintrag in der Datei kommentiert man also mit dem Steuersymbol `#` aus.

```
# Android fstab file.
#<src>                                                  <mnt_point>         <type$
# The filesystem that contains the filesystem checker binary (typically /system) $
# specify MF_CHECK, and must come before any filesystems that do specify MF_CHECK

/devices/soc0/soc.0/2100000.aips-bus/2190000.usdhc/mmc_host* auto auto defaults v$
/devices/soc0/soc.0/2100000.aips-bus/2184200.usb/ci_hdrc.1*  auto auto defaults v$
/devices/soc0/soc.0/2200000.sata/ata1* auto auto defaults voldmanaged=sata:auto

#/dev/block/mmcblk2p2    /system      ext4    rw,barrier=1                        $
/dev/block/mmcblk2p4    /data        ext4    nosuid,nodev,nodiratime,noatime,nomb$
/dev/block/mmcblk2p6    /cache       ext4    nosuid,nodev,nomblk_io_submit       $
/dev/block/zram0 none swap defaults zramsize=314572800
```

Abbildung 25: Änderung der Partitionierungskonfiguration

Allerdings muss sichergestellt werden, dass die eingesteckte SD die übrigen Bedingungen besonders in Bezug auf Data- und Cache-Partition erfüllt. Eine mögliche Partitionierung und zugleich der Standard für die Android-SD-Karte für das Wandboard ist in Abbildung 26 zu sehen.

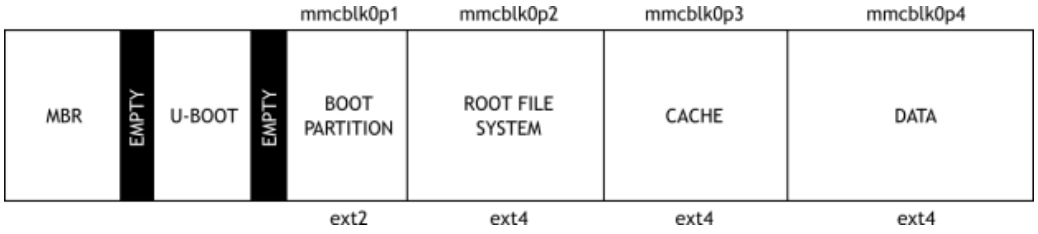


Abbildung 26: Partitionierung der SD-Karte
(http://android.serverbox.ch/wp-content/uploads/2013/06/partition.png)

Dabei sind nach dem Master Boot Record der SD ein acht MiB großer Bereich für den U-Boot-SPL reserviert. Es folgen 33 MiB Bootpartition mit dem U-Boot-Image, eine 1024 MiB große System-Partition, die im Fall eines normalen Bootvorgang das Wurzeldateisystem enthält und eine Cache-Partition in gleicher Größe. Die restliche SD-Karte wird von der Datenpartition eingenommen.
Nach der Überprüfung der Speicherkarte und der U-Boot-Parameter kann mit `run netboot` ein erneuter Boot-Vorgang gestartet werden.
Es gelingt dem Bootloader in diesem Fall die System-Dateien zu laden und die Data- und Cache-Partition zu mounten, auch werden die ersten Services durch den Dienst `init` gestartet. Allerdings bleibt das Hochfahren mit dem Zugriff auf den NFS-Server in einer Fehlermeldung hängen, die vermeldet, dass von diesem keine Antwort erhalten wird.
Für die Lösung dieses Problems müssen zwei Quelldateien des Net-Deamons an NFS angepasst werden. Diese sind im Android-Quellverzeichnis über den

Pfad `system/netd/server/` erreichbar. In der Datei `CommandListener.cpp` ist bereits das Einlesen eines NFS-Modus `ro.nfs.mode` vorgesehen, diese Funktion allein löst die Problemstellung allerdings nicht. Dafür muss die zweite Quelle, die Datei `RouteController.cpp`, ergänzt werden. In deren Standard-Konfiguration erfolgt der Aufruf der Methode `flushRules()`, die in regelmäßigen Abständen aktuelle Routing-Regeln löscht und damit auch die Verbindung zum NFS-Server unterbricht. Die in der Datei enthaltene Funktion `RouteController::Init` ist daher um eine Bedingungsabfrage zur Unterscheidung von normalen und netzbasierten Boot-Vorgängen zu erweitern. Für die if-Anweisung in Abbildung 27 muss die darüber dargestellte Funktion implementiert werden. Dabei ist zu beachten, dass der Import `#include <cutils/properties.h>` im Kopfbereich der Quelldatei existieren muss, um den Aufruf des NFS-Modus zu ermöglichen.

```
static bool isNFSbooted() {
    char prop_value[PROPERTY_VALUE_MAX] = {'\0'};
    if (property_get("ro.nfs.mode", prop_value, "no")) {
        if (strcmp(prop_value, "yes") == 0)
        return true;
    }
    return false;
}

int RouteController::Init(unsigned localNetId) {
    if (!isNFSbooted()){
        if (int ret = flushRules()) {
        return ret;
        }
    }
    if (int ret = addLegacyRouteRules()) {
        return ret;
    }
```

Abbildung 27: Anpassung des Net-Deamons

Mit den abgespeicherten Änderungen ist es nötig, den Quellcode des Deamons neu zu kompilieren. Wie bereits bei der Übersetzung des Android-Projekts wird auch in diesem Fall mit `. build/envsetup.sh` oder mit `source build/envsetup.sh` die Entwicklungsumgebung im Android-Quellverzeichnis gestartet und über die Anweisung `lunch 23` der Gerätetypus `wandboard-eng` gewählt. Daraufhin kann der erneute Bau der Binärdatei `netd` beginnen, der Befehl `mmm -j4 system/netd` steht dafür zur Verfügung. Zuletzt ist es notwendig, die veränderte Datei an die vorgesehene Position im NFS-Verzeichnis zu kopieren und die dortigen Daten zu überschreiben. Das wird mithilfe von `sudo cp out/target/product/wandboard/system/bin/netd /srv/nfsandroid/system/bin/netd`

ermöglicht, sodass nun der geänderte Net-Deamon angewendet wird.
Der letzte Schritt zum reibungslosen Ablauf des Boot-Vorgangs umfasst die Einfügung des Modus `ro.nfs.mode`, der im Route-Controller des `netd` aufgerufen wird, in die Startdatei `init.freescale.rc`, die verschiedene Regeln zum Start verwaltet. Die in Abbildung 28 dargestellte Zeile ist dazu unter der Rubrik `on boot` neu einzutragen.

```
on boot
    # Add NFS property to prevent ip rules flushing
    setprop ro.nfs.mode yes

    # emmulate battery property
    setprop sys.emulated.battery 1
```

Abbildung 28: Ausschnitt aus der Datei `init.freescale.rc`

Mit dem Neustart des Bootloaders erfolgt der Netzwerkboot des Android-Systems nun ohne das Auftreten von Fehlern. Mit dem Erreichen dieses Zieles ist die Methode abgeschlossen, der eigentliche Prozess des Hochfahrens wird im folgenden Ergebniskapitel behandelt.

# 4 Ergebnis

Folgt man dem in der Methodik dargelegten Weg zur Einrichtung des Wandboards, der zugehörigen SD-Karte und der Server, ist es möglich, Android auf der Entwicklungsplattform über das Netzwerk zu laden und zu starten. Dieses Kapitel widmet sich dem Bootvorgang als Ergebnis des mit dieser Arbeit vorgestellten Ansatzes.
Der Netzwerkstart beginnt mit dem Hochfahren des Bootloaders U-Boot, der zuallererst die Bedingungen für die Wahl der Boot-Methode prüft. Wenn dieser auf der SD-Karte die Kernel-Datei `zImage` und das RAM-Disk-Image mit U-Boot-Header `uramdisk.img` findet, erfolgt ein normaler Start von der Speicherkarte, wenn der Countdown abgelaufen ist. Sind diese Dateien nicht vorhanden, wird automatisch ein Netzwerkstart versucht.

Diese Art des Hochfahrens wird über das Laden des Kernels über TFTP initialisiert (Abb. 29). Dazu werden die für diesen Vorgang benötigten Informationen akquiriert, das heißt die eigene und die IP-Adresse des Servers sowie der Name des Kernels und dessen spätere Speicheradresse. Je nach den vorhandenen Internetressourcen kann dieser Ladevorgang nach wenigen Sekunden oder Minuten abgeschlossen sein, abhängig von der Anzahl benötigter Versuche.

```
Booting from net ...
Using FEC device
TFTP from server 192.168.2.106; our IP address is 192.168.2.102
Filename 'zImage'.
Load address: 0x12000000
Loading: #################################################################
         #################################################################
         ########
```

Abbildung 29: Laden des Kernels über TFTP

Ist das Transferieren des Abbilds `zImage` vollständig, lädt der Bootloader den Device Tree Blob für das Wandboard Quad, ebenfalls über TFTP. Es werden dabei die gleichen Datentypen wie beim Laden des Kernels vorbereitet. Aufgrund der kleinen Größe der dtb-Datei ist die Übermittlung innerhalb von Millisekunden beendet.
Anschließend werden die Dateien an die Offsets im Speicher gebracht, von denen sie geladen werden und ihre Funktion vollführen. Das Kernel-Image wird dabei an der Speicherstelle `0x12000000`, der Device Tree Blob an Offset `0x18000000` abgelegt.
Mit diesen Voraussetzungen, zu sehen in Abbildung 30 kann der Kernel gestartet werden und der Boot des Android-Systems beginnt.

```
         #################################
         893.6 KiB/s
done
Bytes transferred = 8141536 (7c3ae0 hex)
Using FEC device
TFTP from server 192.168.2.106; our IP address is 192.168.2.102
Filename 'imx6q-wandboard.dtb'.
Load address: 0x18000000
Loading: ####
         721.7 KiB/s
done
Bytes transferred = 48079 (bbcf hex)
Kernel image @ 0x12000000 [ 0x000000 - 0x7c3ae0 ]
## Flattened Device Tree blob at 18000000
   Booting using the fdt blob at 0x18000000
   Using Device Tree in place at 18000000, end 1800ebce

Starting kernel ...

```

Abbildung 30: Abschluss TFTP-Ladevorgang und Datenausführung

Das Hochfahren des Android-Systems wird begleitet von einer großen Menge an Log-Nachrichten, die über den Start von Diensten oder den Erfolg von Operationen aufklären. In diesem Abschnitt werden einzelne Vorgänge anhand ihres Outputs näher betrachtet, besonders solche, die mit den beschriebenen Konfigurationen in der Methodik in direkter Verbindung stehen.

Unter anderem kann während des Boots das Mounten der Data- und der Cache-Partition verfolgt werden (Abb. 31).

```
fs_mgr: check_fs(): mount(/dev/block/mmcblk2p4,/data,ext4)=0: Success
fs_mgr: check_fs(): unmount(/data) succeeded
fs_mgr: Running /system/bin/e2fsck on /dev/block/mmcblk2p4
e2fsck: e2fsck 1.42.9 (28-Dec-2013)

e2fsck: data: clean, 634/122160 files, 37939/488448 blocks

EXT4-fs (mmcblk2p4): Ignoring removed nomblk_io_submit option
EXT4-fs (mmcblk2p4): mounted filesystem with ordered data mode. Opts: nomblk_io_
submit,noauto_da_alloc,errors=panic
fs_mgr: __mount(source=/dev/block/mmcblk2p4,target=/data,type=ext4)=0
EXT4-fs (mmcblk2p6): Ignoring removed nomblk_io_submit option
EXT4-fs (mmcblk2p6): recovery complete
EXT4-fs (mmcblk2p6): mounted filesystem with ordered data mode. Opts: nomblk_io_
submit,errors=remount-ro
fs_mgr: check_fs(): mount(/dev/block/mmcblk2p6,/cache,ext4)=0: Success
```

Abbildung 31: Einbindung der Data- und Cache-Partition

Die Änderung der Datei `fstab.freescale` und die Anpassung der SD-Kartenpartitionierung ermöglichen die fehlerfreie Durchführung dieses Schrittes.
Im weiteren Verlauf erfolgt die Verbindung zum NFS-Server und somit zum Dateisystem. Dabei kann der Ablauf zum erfolgreichen Mounten der RAM-Disk mit System-Partition nachvollzogen werden.
Nach der Einrichtung der Verbindung über die Schnittstelle `eth0` sendet U-Boot eine DHCP-Anfrage an den Router und erhält die benötigten Informationen über das Netz und den anzusprechenden Server. Die Verknüpfung ist nun etablierbar, die Zugriffsschnittstelle VFS (Virtual File System) bestätigt anschließend die Einbindung von `root (nfs filesystem)`, das heißt das Dateisystem wurde erfolgreich als Root Device beziehungsweise Wurzelverzeichnis gemountet.
Damit sind alle benötigten Android-Daten vorhanden und der Prozess `init`, der die Services initialisiert, wird gestartet. Dieser dominiert den weiteren Bootprozess und vollführt und überwacht den Startprozess der Services. Für alle Vorgänge gibt `init` eine Status- oder im gegensätzlichen Fall eine Fehlermeldung aus. Die meisten Android-Applikationen basieren auf diesen Service-Diensten oder benötigen diese für ihre Funktion, so kann beispielsweise der Start des Media-Dienstes, des Install-Deamons oder des Gatekeepers beobachtet werden. Die Abbildung 32 zeigt einen Ausschnitt des in diesem Absatz beschriebenen Ablaufs.

```
IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
Sending DHCP requests ..., OK
IP-Config: Got DHCP answer from 192.168.2.1, my address is 192.168.2.102
IP-Config: Complete:
     device=eth0, hwaddr=00:1f:7b:b4:1e:41, ipaddr=192.168.2.102, mask=255.255.2
55.0, gw=192.168.2.1
     host=192.168.2.102, domain=speedport.ip, nis-domain=(none)
     bootserver=0.0.0.0, rootserver=192.168.2.106, rootpath=
     nameserver0=192.168.2.1
ALSA device list:
  #0: sgtl5000-audio
  #1: imx-spdif
  #2: imx-hdmi-soc
VFS: Mounted root (nfs filesystem) on device 0:14.
devtmpfs: mounted
Freeing unused kernel memory: 580K (c0d9f000 - c0e30000)
init: init started!
```

Abbildung 32: Etablierung der NFS-Serververbindung

Mit dem Service `bootamin`, kurz für `bootanimation`, der zumeist zuletzt initialisiert wird, fährt schließlich die graphische Nutzeroberfläche hoch. Endet `bootamin` daraufhin wie in Abbildung 33 mit Status 0, ist der Netzwerk-

start beendet und die eigentliche Bedienoberfläche über die HDMI-Anbindung sichtbar und mithilfe einer USB-Maus nutzbar. Die Dauer des gesamten Prozesses hängt erneut von der Netzgeschwindigkeit ab, erfolgt insgesamt aber selbst bei kleinerer Internetressource innerhalb weniger Minuten.

```
init: Starting service 'bootanim'...
warning: `main' uses 32-bit capabilities (legacy support in use)
lowmemorykiller: lowmem_shrink: convert oom_adj to oom_score_adj:
lowmemorykiller: oom_adj 0 => oom_score_adj 0
lowmemorykiller: oom_adj 1 => oom_score_adj 58
lowmemorykiller: oom_adj 2 => oom_score_adj 117
lowmemorykiller: oom_adj 3 => oom_score_adj 176
lowmemorykiller: oom_adj 9 => oom_score_adj 529
lowmemorykiller: oom_adj 15 => oom_score_adj 1000
acc_open
acc_release
init: Starting service 'dhcpcd_eth0'...
init: Service 'bootanim' (pid 928) exited with status 0
```

Abbildung 33: Abschluss Boot-Vorgang

# 5 Diskussion

Mit dieser Arbeit wird eine Methodik vorgestellt, die das Booten von Android-Systemabbildern zur Malware-Analyse auf dem Wandboard erleichtern und verbessern soll. Dazu wurde der Bootloader U-Boot so modifiziert, dass er das Betriebssystem nicht von der SD-Karte, sondern über das Netzwerk lädt und startet.
Dafür war es vor der eigentlichen Konfiguration nötig, die Dateien zu erstellen, die für den alternativen Bootvorgang benötigt werden, da diese nicht in den für das Wandboard bereitgestellten Archiven enthalten sind. Dazu wurden zwei verschiedene Ansätze erarbeitet. Einerseits besteht die Möglichkeit, Android aus den für das Wandboard erstellten Quelldateien zu kompilieren. Dieser Vorgang wurde auf einem Linux-System nach dessen Konfigurierung und der Etablierung einer Übersetzungsumgebung durchgeführt. Dabei werden der Kernel, die Device Tree Blob-Dateien und der Bootloader einzeln erstellt, bevor diese in das eigentliche Betriebssystem, das Android Open Source Project, eingebunden werden, sodass mit Abschluss der Kompilierung ein fertig einsetzbares Android in Form der erforderlichen Dateien zum Start über Netzwerk vorliegen.
Andererseits ist auch die Umwandlung bestehender Dateien aus den vorinstallierten SD-Karten-Images für das Wandboard möglich. Dieser Weg ist im Vergleich mit weniger zeitlichem Aufwand verbunden und kann ebenfalls mit Linux als Host realisiert werden. Da die auf der SD zu findenden Dateien nicht direkt kompatibel verwendbar sind, müssen diese vor dem Einsatz mittels einer je nach Datei unterschiedlichen Befehlsstruktur umgewandelt werden. Besonders wenn bereits Konfigurationen am System auf einer SD-Karte durchgeführt wurden, ist diese Variante sinnvoller, um unnötige Wiederholungen zu verhindern.
Die mit beiden Ansätzen erlangten Daten wurden im zweiten Teil der Methodik auf einem TFTP- und einem NFS-Server eingerichtet. Der erste beinhaltet den Android-Kernel und die Binärdateien der Device Tree Blobs, der zweite die RAM-Disk, das Wurzelverzeichnis und in diesem den Inhalt der System-Partition. Beide Server können auf einem Linux-Host über die Paketverwaltung installiert und über textbasierte Dateien konfiguriert werden. Die Initialisierung der Netzwerkprotokolle und Server ermöglichte daraufhin die Konfiguration und Anpassung des Bootloaders zum Netzwerkstart. Diese erfolgte über die Neudefinition von Adress- und System-Werten sowie die Nutzung der existierenden Umgebungsvariable `netboot`. Mithilfe dieser Funktionalität übernimmt das U-Boot den um NFS erweiterten PXE-Prozess und startet Android auf dem Wandboard über die LAN-Verbindung.

Der mit dieser Ausarbeitung erläuterte Ansatz bringt dem Gesamtprojekt zur Analyse von Android-Malware eine effizientere Verbindung zwischen Webserver und Analyseplattform als die bisherige Methode über die SD-Karte. Dabei entfällt die Notwendigkeit, die Speicherkarte mit dem Wechsel der Android-Version und Malware auszutauschen. Soll beispielsweise ein neuartiger Virus auf einer bereits vorliegenden System-Version analysiert werden, ist es nun ausreichend, einzelne Inhalte des NFS-Servers zu verändern. Selbst der Wechsel der Android-Distribution erfordert ausschließlich den Austausch der in der Methodik beschriebenen System-Dateien auf den Servern. Diese Veränderungen können vollständig digital stattfinden und verringern sowohl den materiellen Aufwand in Bezug auf die Rechnerressourcen und die Nutzung von Speicherkarten als auch den zeitlichen Aufwand in Bezug auf den Arbeitsprozess. Die eingeführte Methode ersetzt den sicheren Start über die SD-Karte mit einer ebenso sicheren Alternativlösung über das Netzwerk und bringt gleichzeitig eine Erhöhung der Effektivität mit sich.

Zur Erreichung des Ziels, Android auf dem Wandboard über alternative und weniger aufwändige Art und Weise zu booten, führen neben dem vorgestellten Weg noch weitere Pfade.
Alternativ zum PXE-Boot zwischen Server und Wandboard besteht auch die Möglichkeit, ein zweites Wandboard einzusetzen, das die Serverfunktionalität übernimmt. Dabei würde die Einrichtung auf einem Android-System über eine App erfolgen, die eine ähnliche Methodik anwendet und der Analyseplattform die zu bootenden Dateien übermittelt. Die Durchführung könnte mit diesem Ansatz auf eine kleinere Hardwareumgebung mit gleichbleibender Leistung gebracht werden. Allerdings existiert das Risiko, dass es durch einzelne Beschränkungen des Betriebssystems und der Funktionsmöglichkeiten der aktuell verfügbaren Apps zu Schwierigkeiten bei der Überführung des Netzwerkstarts vom Rechnersystem auf die Mikrocontrollerplattform kommen kann. Zudem wäre der Kosten-Nutzen-Faktor zu analysieren, da diese Apps zumeist gebührenpflichtig beziehungsweise kostenfrei nur als eingeschränkte Teilversion verfügbar sind.
Weiterhin wäre ein Boot-Vorgang über USB und den USB-OTG-Anschluss denkbar, nach einem ähnlichen Prinzip wie über das Netzwerk, mit zwei Wandboards, die ein Client-Server-Verhältnis bilden und Dateien austauschen. Ein möglicher Ansatz wäre die Nutzung der Android Debug Bridge, die normalerweise zum Senden von Befehlen und Daten, vor allem zu Entwicklungs- und Reparaturzwecken, verwendet wird. Diese Idee ist bereits in einem Projekt erprobt worden, konnte aber keine zufriedenstellenden Ergebnisse erbringen. Die Übermittlung der Boot-Dateien erfolgte über ein USB-OTG-Kabel von der einen Plattform auf die andere, das Hochfahren auf dieser endete

aber mit Fehlermeldungen und konnte abschließend nicht in Funktion gesetzt werden.
Im Vergleich zu den Boot-Methoden über USB kann die Methode über das Netzwerk unter Nutzung von PXE somit als die leichter umsetzbare und zuverlässig funktionale Durchführung betrachtet werden.

Neben der grundlegenden Hauptmethode müssen auch die einzelnen Bestandteile dieser anhand ihrer Vorteile und Alternativen beurteilt werden. Zuallererst ist dabei der verwendete Bootloader U-Boot einzuschätzen. Wie im Methoden- und besonders im Grundlagenkapitel ersichtlich ist, besitzt das U-Boot eine weite Befehls- und Funktionsvielfalt, die es ermöglicht, auf den Speicher zuzugreifen und dessen Elemente zu verändern, verschiedene Boot-Optionen zu wählen und diese den Bedürfnissen des Nutzers anzupassen sowie verschiedene Arten von Skripten auszuführen. Dieser Umfang an Möglichkeiten bringt den Vorteil, dass selbst eher schwierig umzusetzende Projekte in Betrieb genommen werden können, da eventuelle Problemstellungen durch individuell anpassbare Parameter eine Lösung erhalten.
In diesem Fall war der entscheidende Vorzug die bereits existierende Fähigkeit zum Netzwerkboot in Verbindung mit der ebenfalls schon definierten Netboot-Variable, die den programmatischen Ablauf vorgibt. Dadurch mussten nur die IP-Adressen und die Namen der zum Booten benötigten Dateien und Images eingetragen werden. Der Boot-Prozess war daraufhin auch aufgrund der eingebauten Unterstützung der beteiligten Netzwerkprotokolle ohne weitere Änderungen am Universal-Bootloader startbereit.
Das U-Boot wird neben zahlreichen anderen Einsatzgebieten im Bereich der Embedded Systems häufig für die Verbindung Wandboard und Android eingesetzt. Der Programmentwurf des Loaders ist auf die reibungslose Funktion von Android auf dem Wandboard abgestimmt und übernimmt neben den genannten speziellen Aufgaben auch das normale Hochfahren von der SD-Karte. Die Quelldateien werden zusammen mit den Android-Quellarchiven ausgeliefert und die Anwesenheit des erfolgreich kompilierten Bootloaders ist Voraussetzung für die Übersetzung und den Bau des Betriebssystems, dem Android Open Source Project. Somit ist das U-Boot nicht nur eine gut geeignete Umgebung für den Netzwerkstart, sondern auch ein wichtiger Bestandteil für die Funktion des Android-Systems auf dem Wandboard.
Für Mikrocontroller-Plattformen existieren mehrere Arten von Bootloadern, die zwar vereinzelt eine ähnliche Funktionsbreite wie U-Boot aufweisen, aber nicht die gleiche Systemunterstützung und Gerätekompatibilität besitzen. So unterstützt beispielsweise „RedBoot“ zwar das Laden von Abbildern über TFTP, ist allerdings mit der Wandboard-Architektur ARM Cortex-A9 nicht kompatibel. Die Unterstützung des Boots von Android ist ebenfalls nicht er-

sichtlich. Eine sinnvolle Alternative, die U-Boot ersetzen könnte, kann auch nach dem Studium anderer Bootloader nicht empfohlen werden.

In der vorgestellten Methode wird das Protokoll PXE durch das Network File System ergänzt. Der Server stellt das Android-Dateisystem über das lokale Netz zur Verfügung und ermöglicht die Abspeicherung von Änderungen, die im laufenden Betrieb gemacht werden. Das bringt einen entscheidenden Vorteil im Vergleich zum reinen TFTP-basierten Hochfahren, das bei jedem Neustart das gleiche Image bootet. In diesem Fall sind Veränderungen nur temporär und sind beim Abschalten der Client-Hardware wieder verschwunden. Ein Gerät, das im späteren Gesamtprojekt zuverlässig Malware analysieren soll, benötigt allerdings ein stabiles Dateisystem, in der die Voraussetzungen, das heißt die grundlegenden Applikationen und Analyse-Tools, bei jedem Durchgang in gleicher Ausführung vorhanden sind. Der NFS-Server stellt diese Funktionalitäten zur Verfügung.
Ein weiterer Vorteil besteht in der Kompatibilität zwischen U-Boot und NFS. Der Bootloader bindet das NFS-Dateisystem während des Bootvorganges ein, nachdem der TFTP-Ladevorgang abgeschlossen ist. Damit lässt sich die Übermittlung der Systemdateien auf einfache Art und Weise einrichten und durchführen. Außer einigen Anpassungen an Android-Dateien und Quelldateien von Diensten muss ausschließlich der Server und die Freigabetabelle existieren, um das Dateisystem über den Bootloader von einem physisch getrennten Medium zu laden.

Diese Arbeit ergänzt die vorangehenden Projekte zur Konfiguration des Wandboards für die Durchführung einer Malware-Analyse um eine effektivere Art des Bootens und Ladens von Systeminhalten und ermöglicht dadurch die Umsetzung des gesamten Projektes als Webservice. Mit den bisherigen Vorgehensweisen und Ansätzen, das heißt dem SD-Karten- oder USB-Start, wäre die Realisierung nur in umständlicher Form oder überhaupt nicht möglich gewesen.
Daher ist nun im weiteren Forschungsverlauf die Zusammenstellung der Server- und Wandboard-Infrastruktur vorgesehen, dafür werden die zwei Arbeiten (Mo16) und (Wi16) mit dem vorliegenden Projekt verknüpft, um einen einsatzfähigen Laufzeit-Service zur Analyse von Android-Malware auf mehreren Wandboards zu konstruieren.
Der vorgestellte Ansatz besitzt allerdings auch Potenzial zur Weiterentwicklung. So können bestimmte Abläufe wie die Vorbereitung der Server oder der Tauschprozess der Android-Dateien, die aktuell noch per Hand erfolgen, über Skripte automatisiert werden. Auch ist die Berücksichtigung der aktuell funktionsfähigen und die Erforschung neu entwickelter Alternativen wichtig,

um in Zukunft einen möglicherweise noch effektiveren und schnelleren Ansatz hervorzubringen. Von der stetigen Forschung am Projekt und seinen technischen Bestandteilen sowie der Erweiterung des aktuellen Hard- und Software-Stands hängt der fortlaufende Erfolg der Einrichtung ab.

Diese Arbeit schließt die Planung eines Projektes zur Malware-Analyse ab und ermöglicht damit dessen Durchführung. Das Bedürfnis nach der Kenntnis des Vorgehens von digitalen Schädlingen und der damit ermöglichten Klassifizierung der zuvor unbekannten Malware kann beantwortet werden, um für diese schnellstmöglich Maßnahmen entwickeln zu können.
Durch die aktuelle Entwicklung von Straftaten im Bereich der Cyberkriminalität, besonders bei Android-Malware, ist die Bereitstellung eines sicheren, zuverlässigen und schnellen Analyseservices essentiell wichtig, um mit der fortschreitend steigenden Zahl an mobilen Malware-Arten, vor allem bei Viren, mithalten zu können und Mittel zur Bekämpfung und Abwehr der Schadsoftware herzustellen, bevor diese sich unkontrolliert ausbreitet und bei einer großen Anzahl von Nutzern Schaden anrichten kann.

# Quellenverzeichnis

## Literaturquellen

**(Ca00)** B. Callaghan: NFS illustrated. Reading, USA: Addison Wesley Longman, 2000.

**(Ha11)** M. Haßelberg: Embedded Linux in der Mikrocontrollerpraxis. Aachen: Elektor, 2011.

**(Mo16)** D. Morgado Anglés: Konfigurieren einer Hardwareplattform zur Analyse von Android-Schadsoftware. Hochschule Mittweida, Fakultät Angewandte Computer- und Biowissenschaften, Bachelorarbeit, 2016.

**(Wi16)** H.Winkler: Projektstudie: Android-Debugging auf dem Wandboard – Entwicklung einer Plattform zur Analyse von Prozessen in mobiler Malware. Hochschule Mittweida, Institut für Technologie- und Wissenstransfer (ITWM) in Kooperation mit der Fakultät CB, Masterarbeit, 2016.

## Online-Quellen

**(An17)** Android Open Source Project (2017): Downloading and Building - Establishing a Build Environment, erreichbar unter:
https://source.android.com/source/initializing
(zuletzt besucht am: 28.05.2017, 16:28 Uhr).

**(De17)** DENX Software Engineering (2017): U-Boot Design Principles - The 10 Golden Rules of U-Boot design, erreichbar unter:
http://www.denx.de/wiki/U-Boot/DesignPrinciples
(zuletzt besucht am: 01.05.2017, 11:52 Uhr).

**(DM17)** DENX Software Engineering (2017): The DENX U-Boot and Linux Guide (DULG) for canyonlands, erreichbar unter:
http://www.denx.de/wiki/DULG/Manual
(zuletzt besucht am: 24.04.2017, 11:15 Uhr).

**(Ga17)** Gartner (2017): Gartner Says Worldwide Sales of Smartphones Grew 7 Percent in the Fourth Quarter of 2016, erreichbar unter:
http://www.gartner.com/newsroom/id/3609817
(zuletzt besucht am: 20.03.2017, 11:40 Uhr).

**(Ie81)** Internet Engineering Task Force (1981): RFC 783. The TFTP Protocol (Revision 2) - obsoleted, erreichbar unter:
https://tools.ietf.org/pdf/rfc783.pdf
(zuletzt besucht am: 19.06.2017, 16:10 Uhr)

**(Ie92)** Internet Engineering Task Force (1992): RFC 1350. The TFTP Protocol (Revision 2), erreichbar unter:
https://tools.ietf.org/pdf/rfc1350.pdf
(zuletzt besucht am: 21.06.2017, 18:48 Uhr)

**(Ie95)** Internet Engineering Task Force (1995): RFC 1813. NFS Version 3 Protocol, erreichbar unter:
https://tools.ietf.org/pdf/rfc1813.pdf
(zuletzt besucht am: 26.06.2017, 19:25 Uhr)

**(In99)** Intel Corporation (1999): Preboot Execution Environment (PXE) Specification. Version 2.1, erreichbar unter:
http://download.intel.com/design/archives/wfm/downloads/pxespec.pdf
(zuletzt besucht am: 27.04.2017, 18:50 Uhr)

**(NC16)** Nexus Computing GmbH Switzerland, Manuel Di Cerbo: Using Android in Industrial Automation. Building and Deploying Android AOSP 6.01 for the Wandboard, erreichbar unter:
http://android.serverbox.ch/?p=1738
(zuletzt besucht am: 08.05.2017, 11:37 Uhr)

**(NR16)** Nexus Computing GmbH Switzerland, Andreas Rudolf: Using Android in Industrial Automation. Android 6 Marshmallow over Network File System (NFS) on the Wandboard, erreichbar unter:
http://android.serverbox.ch/?p=1777
(zuletzt besucht am: 23.06.2017, 10:55 Uhr)

**(Wa17)** Wandboard.org (2017): Downloads, erreichbar unter:
http://wandboard.org/index.php/downloads
(zuletzt besucht am: 07.05.2017, 11:51 Uhr)

# Anlagen

## Ordnerstruktur der Projekt-CD

In diesem Abschnitt soll die Struktur der beiliegenden CD darstellt werden, die sowohl diese Arbeit in digitaler Form als auch die Dateien der Server und des Bootloaders U-Boot enthält. Bei den Systemdateien handelt es sich um die Android-Version 6.0, an der die vorgestellte Methodik erprobt wurde.

`\Bachelorarbeit` enthält die Bachelorarbeit im PDF-Format.

`\U-Boot-Img` beinhaltet den Secondary Program Loader (SPL) und das für den Netzwerkstart konfigurierte Abbild des Bootloaders U-Boot (uboot.img).

`\Server_TFTP` beinhaltet das Kernel-Abbild (zImage) und die Device Tree Blob-Datei (imx6q-wandboard.dtb), die auf dem Server im TFTP-Verzeichnis platziert sind.

`\Server_NFS` enthält das RAM-Disk-Image (ramdisk.img) sowie das Abbild der System-Partition (system_raw.img), die in den Ordner des NFS-Servers eingebunden beziehungsweise entpackt werden müssen.

# Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Mittweida, den 14.07.2017

Christopher Lenk